autentia

SOFTWARE DESIGN

Principios y patrones del desarrollo de software

Primera parte



GUIA PARA
DIRECTIVOS Y TÉCNICOS

V.1

Software Design

Principios y patrones del desarrollo de software

Este documento forma parte de las guías de onboarding de Autentia. Si te apasiona el desarrollo de software de calidad ayúdanos a difundirlas y anímate a unirte al equipo. Este es un documento vivo y puedes encontrar la última versión, así como el resto de partes que completan este documento, en nuestra web.

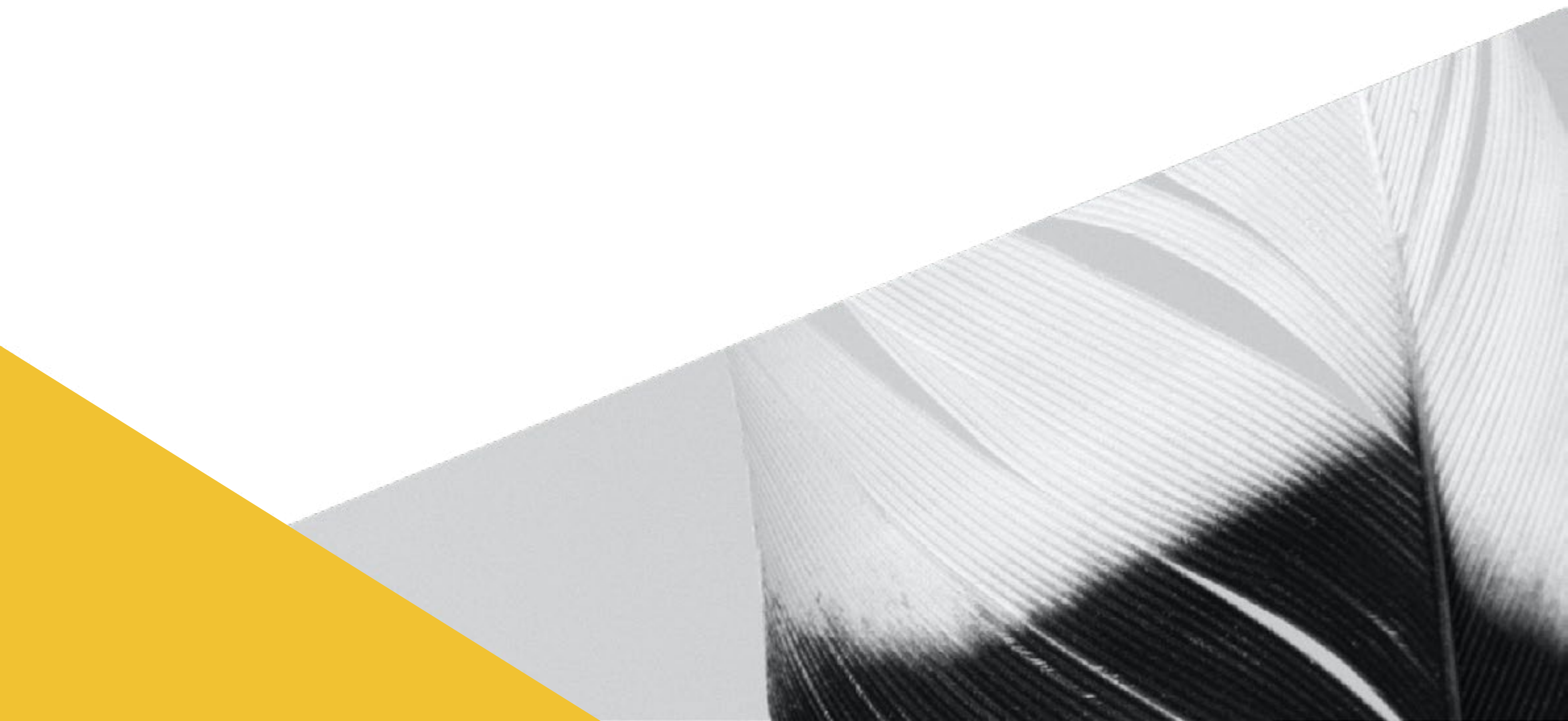
https://www.autentia.com/libros/



Esta obra está licenciada bajo la licencia Creative Commons Attribution ShareAlike 4.0 International (CC BY-SA 4.0)

En las fases iniciales del desarrollo de un proyecto de software, durante sus primeras versiones, las nuevas funcionalidades fluyen de manera graciosa y natural, casi parecen construirse solas y “todo son vino y rosas”. A medida que los proyectos van avanzando y creciendo, de repente, toda esa magia desaparece y comienzan a surgir una serie de problemas y situaciones que muestran que algo no va bien, son síntomas de que nuestros diseños se están degradando y pudriendo.

“Sólo el tiempo demostrará si nuestro software, al enfrentarse a los fuertes vientos del cambio, está asentado sobre roca firme o sobre arena”



Los síntomas más evidentes son:

- **Rigidez:** el software se vuelve difícil de cambiar incluso en tareas sencillas. Las estimaciones son cada vez más abultadas y cada funcionalidad nueva cuesta horrores cuando antes era casi inmediata

- **Fragilidad:** muy relacionado con la rigidez, la fragilidad es la tendencia a que el software se rompa en múltiples sitios cada vez que se hace un cambio, incluso en partes que conceptualmente no tienen relación ninguna las unas con las otras. Cuando subimos una nueva versión que ha costado "sudor y lágrimas", resulta que se rompen cosas que aparentemente no tienen nada que ver con lo que hemos hecho.

- **Inmovilidad o poca reutilización:** cuando resulta imposible reutilizar software de otros proyectos o incluso de otras partes del mismo proyecto. Suele ocurrir porque el módulo que queremos reutilizar tiene una mochila de dependencias demasiado grande como para asumir el esfuerzo y el riesgo de desacoplarlo. Se piden cosas que son prácticamente una copia de otras funcionalidades de las que ya disponemos, sin embargo esto no parece ser una ventaja y cuesta demasiado sacar lo común para reutilizarlo, tendiendose a copiar y duplicar código.

- **Viscosidad:** la viscosidad en el ámbito del diseño se da cuando es más sencillo hacer las cosas mal, hacer la "ñapa", que tratar de

hacerlas por el camino trazado. La viscosidad en el entorno ocurre cuando el ecosistema de desarrollo es lento e ineficiente en el más amplio sentido de la palabra. Por ejemplo, cuando una vez tenemos la funcionalidad terminada, hacerla llegar hasta producción supone una auténtica aventura y se necesitan días (o semanas).



Design Smells autentia

¿Qué son?

Los "olores de diseño" dentro del desarrollo del software son una manera de definir los síntomas que surgen con el avance del proyecto y crecimiento del software que influyen en la degradación del diseño.



CARACTERÍSTICAS

Síntoma	Definición	Consecuencias
Rigidez	Dificultad de realizar cambios en el software, incluso en tareas sencillas.	• Las estimaciones son más abultadas. • Añadir funcionalidades nuevas cuesta mucho cuando antes era más rápido.
Fragilidad	Tendencia a que el software se rompa en múltiples sitios cada vez que se realiza un cambio en él, incluso sin tener relación el cambio con lo que se rompe.	• Subir una nueva versión y que se rompan partes que aparentemente no tienen que ver con lo modificado. • Aparición de mayor cantidad de bugs, aumentando el coste de esfuerzo y tiempo en corregirlos.
Inmovilidad	Incapacidad de reutilizar piezas de código por el gran acoplamiento que tiene, haciendo que este sea inamovible cuando la funcionalidad es prácticamente la misma que la deseada.	• Aumenta la complejidad del diseño y del código al introducir código duplicado. • Complica el entendimiento del código y genera equivocaciones.
Viscosidad	La tendencia de que, en el ámbito del software, realizar la solución buena requiere mayor esfuerzo comparado con la solución mala y rápida y, en el ámbito del entorno de desarrollo, éste es lento e ineficiente.	• Provoca el efecto bola de nieve: cuanto más tarde se corrija esa deuda técnica más difícil y costoso será resolverlo. • Pérdida de tiempo, estrés, confusión, pasotismo, acciones inseguras, etc.

Los principios de desarrollo de software son una serie de **reglas y recomendaciones** específicas que los programadores deben seguir durante el desarrollo si quieren escribir un código limpio, comprensible y fácil de mantener. No hay una varita mágica por medio de la cual se pueda transformar una combinación de variables, clases y funciones en el código ideal, pero hay algunos consejos y sugerencias que pueden ayudar al programador a determinar si está haciendo las cosas bien y tratar de evitar las situaciones que a modo de ejemplo hemos narrado en el apartado anterior, y que seguro, que si llevamos el tiempo suficiente dedicados a esta maravillosa profesión del desarrollo de software, habremos vivido, sino igual, al menos de manera similar.

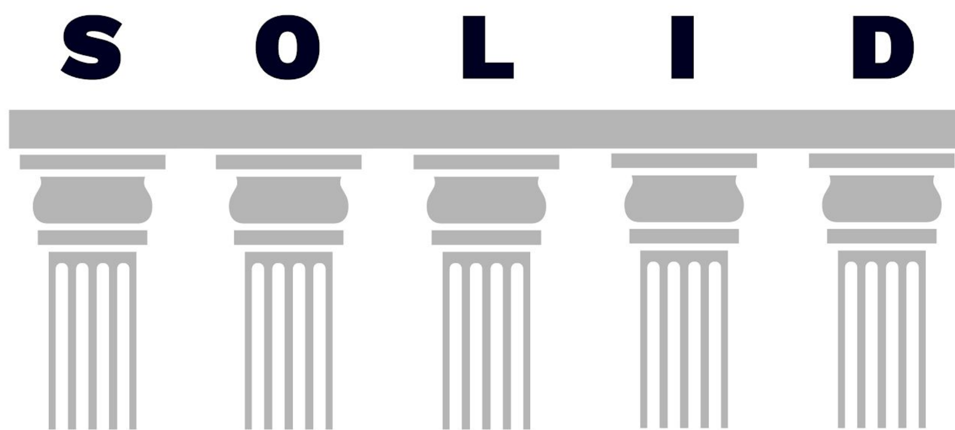
Software Design

Principios y patrones del desarrollo de software

Índice

Principios generales

Principios S.O.L.I.D.



S.O.L.I.D. es un acrónimo mnemónico para cinco principios de diseño destinados a hacer que los diseños de software sean más comprensibles, flexibles y mantenibles. Los principios son un subconjunto de muchos principios promovidos por el ingeniero e instructor de software estadounidense Robert C. Martin. Aunque se aplican a cualquier diseño orientado a objetos, los principios SOLID también pueden formar una filosofía central para metodologías como el desarrollo ágil o el desarrollo de software adaptativo.

Los principios comprendidos en S.O.L.I.D. son:

- **S**: Single responsibility.
- **O**: Open/closed.
- **L**: Liskov substitution.

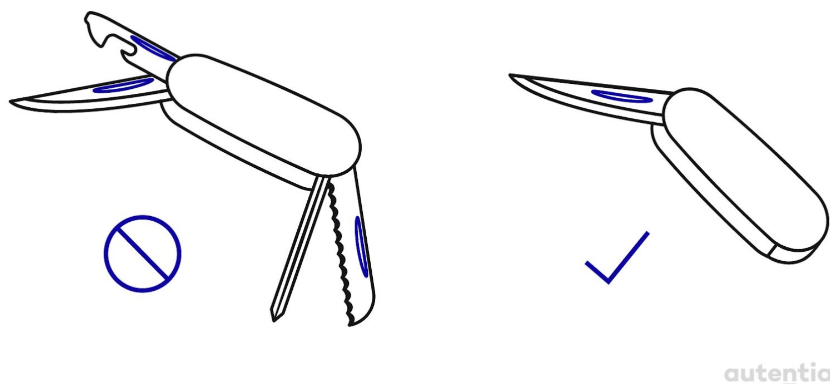
- **I**: Interface segregation.
- **D**: Dependency inversion.

Aplicar estos principios facilitará mucho el trabajo, tanto propio como ajeno (es muy probable que nuestro código lo acaben leyendo muchos otros desarrolladores a lo largo de su ciclo de vida). Algunas de las ventajas de aplicarlo son:

- Facilitar el mantenimiento del código.
- Reducir la complejidad de añadir nuevas funcionalidades.
- Aumentar la reusabilidad de piezas y componentes.
- Mejorar la calidad del código y su comprensión.

Single responsibility (SRP)

El principio de responsabilidad única o single responsibility establece que **un módulo de software debe tener una y solo una razón para cambiar**. Esta razón para cambiar es lo que se entiende por responsabilidad.



“Reúna las cosas que cambian por las mismas razones. Separe las cosas que cambian por diferentes razones.”

Este principio está estrechamente relacionado con los conceptos de acoplamiento y cohesión. Queremos aumentar la cohesión entre las cosas que cambian por las mismas razones y disminuir el acoplamiento entre las cosas que cambian por diferentes razones. Este principio trata sobre **limitar el impacto de un cambio**.

Si existe más de una razón para cambiar una clase, probablemente tenga más de una responsabilidad. Otro posible "mal olor" es que tenga diferentes comportamientos dependiendo de su estado. Tener más de una responsabilidad también hace que el código sea difícil de leer, testear y mantener. Es decir, hace que el código sea menos flexible.

Entre las ventajas de aplicar este principio encontramos que, si se necesita hacer algún cambio, éste será fácil de detectar ya que estará aislado en una clase claramente definida y comprensible. Minimizando los efectos colaterales en otras clases. Algunos ejemplos que encontramos en la vida real son:

Si cambia la forma en que se compra un artículo, no tendremos que modificar el código responsable de almacenarlo. Si cambia la base de datos, no habrá que arreglar cada pedazo de código donde se utiliza.

Para más detalle, ver el artículo *[S.O.L.I.D.] Single responsibility principle / Principio de Responsabilidad Única* en Adictos al Trabajo.

SOLID - Single Responsibility Principle autentia

Una clase debe tener solo una razón para cambiar

El **Single Responsibility Principle (SRP)** o principio de responsabilidad única es un principio que defiende que una clase o módulo debería tener responsabilidad sobre una única funcionalidad del software.

CONCEPTO

Robert C. Martin define el principio con la siguiente frase: ***"Una clase debe tener solo una razón para cambiar."***

¿Pero qué entendemos como **"razón"**?

Robert aclara que el principio trata realmente sobre las **personas**. No queremos que una misma pieza de código tenga que cambiar debido a personas distintas ya que las personas son las que dirigen los cambios en el software.

Por ejemplo, un cambio en *Employee* sobre su forma de trabajar vendrá requerido por una persona distinta del que vendrá un cambio sobre la tecnología de persistencia de datos.

Esto podría entenderse como **razones** distintas para cambiar.

EJEMPLO

Podemos identificar que la clase Employee tiene dos responsabilidades, la propia de un empleado que es trabajar y la de persistir su estado.

Podemos separar ambas responsabilidades en dos clases:

Employee
+ work(): void
+ save(): void

Employee
+ work(); void

EmployeeRepository
+ save(Employee): void

Open/closed (OCP)

Este principio nos dice que **los módulos de software deben ser abiertos para su extensión pero cerrados para su modificación**. ¿A qué se refiere con esto?

- **Abierto para la extensión**: esto significa que el comportamiento del módulo puede extenderse. A medida que cambian los requisitos de la aplicación, podemos ampliar el módulo con nuevos comportamientos que satisfagan esos cambios. En otras palabras, podemos cambiar lo que hace el módulo.
- **Cerrado por modificación**: un módulo estará cerrado si dispone de una descripción (interface) estable y bien definida. Extender el comportamiento de un módulo no debería afectar al código ya existente en el módulo, es decir, el código original del módulo no

debería verse afectado y tener que modificarse.

Y esa es realmente la esencia de este principio. Debería ser fácil cambiar el comportamiento de un módulo sin cambiar el código fuente de ese módulo. Esto no significa que nunca cambiará el código fuente. Lo que significa es que debemos esforzarnos por lograr que nuestro código esté estructurado de forma que, cuando el comportamiento cambie de la manera esperada, no debamos hacer cambios radicales en todos los módulos del sistema. Idealmente, podremos agregar el nuevo comportamiento, añadiendo código nuevo y cambiando poco o nada del código antiguo.

La forma de implementar este principio en el mundo práctico, es a través del polimorfismo, ya sea por interfaces o clases abstractas.

Para más detalle, ver el artículo *[S.O.L.I.D.] Open-Closed Principle / Principio Abierto-Cerrado* en Adictos al Trabajo.

SOLID - Principio Open/Closed autentia

CLOSED

Abierto a la extensión, cerrado a la modificación

Representa la O de SOLID. Con este principio se pretende minimizar el efecto cascada que puede suceder cuando cambiamos el comportamiento de una clase. Si existen clientes que dependan de ella, es posible que tengan que cambiar su comportamiento también.

PLANTEAMIENTO

El principio consta de dos partes:

- Las clases deben estar **abiertas** a la extensión. La extensión se refiere a las modificaciones que pueden ocurrir cuando se plantean nuevos requisitos de software.
- Las clases deben estar **cerradas** a la modificación. No se puede cambiar el código fuente de una clase.

El polimorfismo es la herramienta principal para unir estos dos conceptos que a primera vista parecen contradecirse.

Este principio nos proporciona una mayor estabilidad de los clientes que dependan de la antigua clase. La antigua clase ya funciona, está probado y demostrado. Si incorporamos nuevas funcionalidades a ella, aumentamos la posibilidad de introducir bugs en el software. Por ello, cada vez que se quiera añadir o modificar un comportamiento, en vez de cambiar el código existente, se extiende la clase abstracta o la interfaz y se realizan los cambios en una nueva clase.

EJEMPLO

En una oficina se encuentra una cafetera utilizada por varias personas. Un día, alguien pidió que el café se pudiera preparar distinto. Esta solicitud implicaba un cambio en la configuración y comportamiento de la cafetera.

Persona1
Persona2
CafeteraSencilla
Persona3
Quiero mi café de otra manera

En vez de cambiar el comportamiento de la cafetera actual para intentar satisfacer a dos clientes distintos, aplicaremos el principio Open/Closed. Cerramos nuestra clase CafeteraSencilla ante cualquier modificación, pero creamos una interfaz para abrirla a extensiones:

<<Interface>>
Cafetera
Persona1
Persona2
CafeteraSencilla
CafeteraConEspuma
Persona3

Logramos cumplir con el nuevo requisito sin tener que arriesgar los anteriores. Es más, ahora ofrecemos la posibilidad de utilizar una cafetera o la otra, en caso de que cambien de opinión.

Liskov substitution (LSP)

“Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta.”

La sustitución de Liskov nos dice que los **objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento** del programa. Básicamente, si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la clase padre.

Este principio nos **ayuda a utilizar la herencia de forma correcta** y nos muestra que no se debe mapear automáticamente el mundo real en un modelo orientado a objetos, ya que no existe una equivalencia unívoca entre ambos modelos.

Para más detalle, ver el artículo *[S.O.L.I.D.] Liskov substitution* en Adictos al Trabajo.

SOLID - Liskov Substitution Principle autentia

Definición

Concepto introducido por Barbara Liskov que representa la L de los principios S.O.L.I.D. El principio dice: **una clase que hereda de otra debe poder usarse como su padre sin necesidad de conocer las diferencias entre ellas**.

¿EN QUÉ CONSISTE?

Este principio nos ayuda a utilizar correctamente la herencia ya que **los objetos de nuestras subclases se deben comportar de igual forma que los de la superclase**.

Imaginemos que tenemos una clase que hereda de otra, pero hay un método que no se necesita o no se usa en esa clase. Para solucionar esto, podriamos devolver null o una excepción en ese método. Al hacer esto, estamos violando el Principio de Sustitución de Liskov. Si el método de la clase original no lanza ninguna excepción, los métodos sobrescritos de las subclases tampoco deberían hacerlo. O, si estamos heredando de clases abstractas que nos obligan a devolver null o lanzan una excepción, también estaríamos ante una violación del principio.

Hay una frase muy conocida que dice 'Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta'.

DISEÑO POR CONTRATO

Para cumplir con el principio, Liskov propuso un concepto parecido al *diseño por contrato (design by contract).*

Cuando se llame a un método de la subclase, estos deben cumplir una serie de precondiciones y postcondiciones. Las precondiciones deben ser verdaderas para que el método se pueda ejecutar. Una vez se ha ejecutado, las postcondiciones deben ser verdaderas también.

Se establecieron ciertas restricciones sobre cómo el contrato de una superclase podía seguir ciertos patrones que permitiese aplicar la herencia correctamente.

- Las **precondiciones** no pueden ser **más restrictivas** en un subtipo.
- Las **postcondiciones** no pueden ser **menos restrictivas** en un subtipo.
- Las **invariantes** establecidas por el supertipo deben ser mantenidas por los subtipos.

Interface segregation (ISP)

El principio de segregación de interfaces establece que **muchas interfaces cliente específicas son mejores que una interfaz de propósito general**. Cuando los clientes son forzados a utilizar interfaces que no usan por completo, están sujetos a cambios de esa interfaz. Esto al final resulta en un acoplamiento innecesario entre los clientes.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz, cuya funcionalidad este cliente no usa pero que otros clientes si, este cliente estará siendo afectado por los cambios que fuercen otros clientes en la clase en cuestión.

Debemos intentar evitar este tipo de acoplamiento cuando sea posible. Esto se consigue separando las interfaces en otras más pequeñas y específicas.

Para más detalle, ver el artículo *[S.O.L.I.D.] Interface Segregation Principle / Principio de segregación de interfaz* en Adictos al Trabajo.

SOLID - Interface Segregation Principle autentia

¿Cómo lo aplico?

El **interface Segregation Principle (ISP)** o principio de segregación de interfaces defiende que ninguna clase debería depender de métodos que no usa.

CONCEPTO

Cuando creamos una interfaz debemos estar seguros de que la clase que va a implementar la interfaz va a poder implementar todos los métodos.

En caso contrario **debemos separar la interfaz en interfaces más pequeñas** con menos métodos.

A veces, anticipar qué métodos va realmente a necesitar las implementaciones no es sencillo. Si hemos aplicado correctamente el principio de responsabilidad única y el principio de sustitución de Liskov podremos tener una buena aproximación.

Por lo general siempre será preferible **muchas interfaces pequeñas** a una gran interfaz con muchos métodos.

EJEMPLO

En el ejemplo podemos observar que un Atún se ve forzado a tener un método volar, el cual no puede implementar, ya que un atún no puedo volar.

Podríamos separar ambas habilidades en distintas interfaces. Una para voladores y otra para nadadores.

De este modo los animales implementarán solo las interfaces que necesiten.

«interface» Animal
+ swim()
+ fly()
Duck
+ swim()
+ fly()
+ quack()
Tuna
+ swim()
+ fly()
«interface» Flying
+ fly()
«interface» Swimmer
+ swim()
Duck
+ swim()
+ fly()
+ quack()
Tuna
+ swim()

Dependency inversion (DIP)

El principio de inversión de dependencia nos dice que **las entidades de software deben depender de abstracciones, no de implementaciones**. A su vez, los módulos de alto nivel no deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.

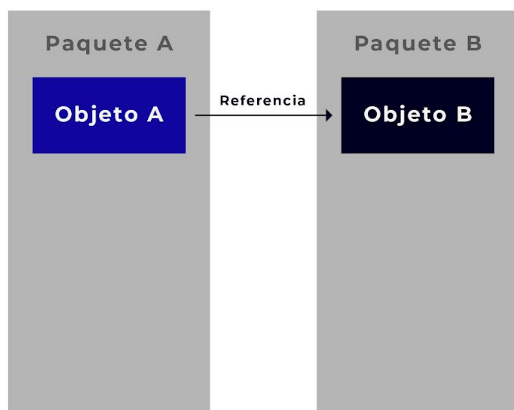


Figura 1

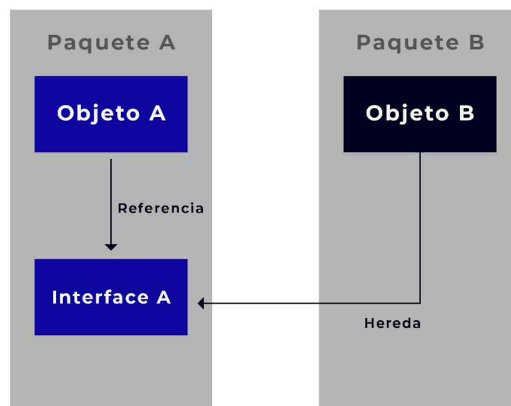


Figura 2

autentia

Mediante este principio ocultamos los detalles de implementación, ganando en flexibilidad. Cuando estamos haciendo tests, podemos reemplazar dependencias reales por objetos mockeados.

Gracias a esta flexibilidad, vamos a poder sustituir componentes sin que los clientes que los consumen se vean afectados ya que dependen de la abstracción y no de la implementación concreta.

Lo que se pretende es que no existan dependencias directas entre módulos, sino que dependan de abstracciones. De esta forma, nuestros módulos pueden ser más fácilmente reutilizables. Es fundamental que la abstracción se defina en base a las necesidades del módulo o cliente y no en las capacidades de la implementación, de lo contrario, la abstracción estaría bastante acoplada a la implementación teniendo así menos flexibilidad.

Para más detalle, ver el artículo *[S.O.L.I.D.] Dependency inversion principle / Principio de inversión de dependencias* en Adictos al Trabajo.

SOLID - Dependency Inversion autentia

¿Qué es?

Representa la D de los principios S.O.L.I.D. El principio dice, los **módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones**. El principio tiene como fin evitar depender de concreciones para minimizar el grado de acoplamiento entre los componentes.

PROBLEMA - SOLUCIÓN

Normalmente, las capas de alto nivel (ClassA) consumen las de bajo nivel (ClassC), generando un fuerte acoplamiento.

ClassA → ClassB → ClassC

Para evitar este problema, se introduce una capa de abstracción que permite reutilizar las capas de mayor nivel.

ClassA → <<Interface>> ClassB

ClassB → <<Interface>> ClassC

ClassC

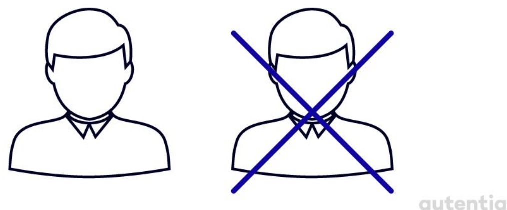
VENTAJAS

Si aplicamos correctamente los anteriores principios SOLID como el Open/Closed y el de Liskov, estamos implícitamente cumpliendo con la inversión de dependencias. Pero, ¿qué ventajas nos aporta?

- **Mayor flexibilidad haciendo testing**: gracias a que se depende de abstracciones, podemos reemplazar objetos reales por mocks que simulen el comportamiento deseado.
- **Reduce el acoplamiento** entre clases: al no depender de una implementación en concreto, no acoplamos nuestro código a ciertas clases específicas.
- Código más **limpio, legible y mantenible** en el tiempo: al no depender de implementaciones, estamos contribuyendo a un desarrollo más robusto y que no sea frágil a cualquier cambio.
- Al depender de abstracciones, tenemos la **posibilidad de elegir en tiempo de ejecución la implementación** adecuada.

Don’t Repeat Yourself (DRY)

DRY: Don’t repeat Yourself



Su objetivo principal es **evitar la duplicación de lógica**. Cada pieza de funcionalidad debe tener una **identidad única**, **no ambigua** y **representativa** dentro del sistema.

Según este principio toda pieza de funcionalidad nunca debería estar duplicada ya que esta duplicidad incrementa la dificultad en los cambios y

su evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias.

Por pieza de funcionalidad, no nos referimos a código sino a lógica, y más concretamente a función lógica. No es saludable tener tres métodos para abrir una conexión a una base de datos, cada uno con su propio código, si no hay un motivo que así lo justifique. Como se ve en este caso, los tres métodos pueden tener distinto código, pero la función final de los tres sigue siendo conectarse a la base de datos. Si en la evolución de nuestro software se nos solicita cambiar la forma de conectarnos, el tipo de base de datos o incluso enviar los datos a otros sistema de almacenamiento, deberemos modificar los tres métodos, lo que incrementa la cantidad de trabajo ya que debemos escribir el código tres veces y, consecuentemente, probarlo, introduce más posibilidades de cometer errores, aumenta la complejidad del código y más. Si además, no somos los autores originales de todo este código y solo nos toca mantenerlo, todo parece más difícil y laborioso aún.

Cuando el principio DRY se aplica de forma eficiente, los cambios en cualquier parte del proceso requieren cambios en un único lugar. Por el contrario, si algunas partes del proceso están repetidas por varios sitios, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece no se encuentran sincronizados.

En resumen, ¿por qué es importante?

- **Hace el código más mantenible**. Evitar la repetición de lógica permite que si alguna vez cambia la funcionalidad en cuestión, no lo tengas que hacer en todos los lugares en los que lo repetiste.
- **Reduce el tamaño del código**. Esto lo hace más legible y entendible porque hay menos código.
- **Ahorra tiempo**. Al tener pedazos de lógica disponibles para reutilizarlos, en el futuro, estamos más preparados para lograr lo

mismo en menos tiempo.

Principio DRY autentia

¿Qué es?

El principio DRY es un acrónimo del inglés “Don’t Repeat Yourself” que significa “No te repitas” y éste consiste en evitar las duplicaciones lógicas en el software.

DESTACAR

Evitar duplicaciones lógicas

≠

Evitar duplicaciones de código

OBJETIVOS

Cada pieza de funcionalidad lógica debe tener dentro del sistema:

- **Una identidad única.**
- **No ambigua.**
- **Representativa.**

EJEMPLO

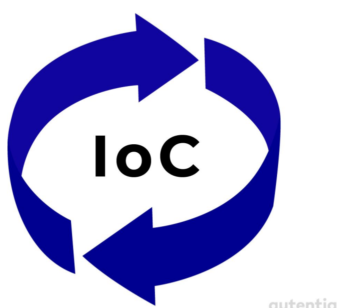
1. Imagina tener tres métodos para abrir una conexión a una base de datos. Cada uno tiene su propio código, pero la función final es conectarse a la base de datos.
2. Si más adelante, cambia la manera de conectarse a la base de datos, el tipo de base de datos o enviar datos a otros sistemas de almacenamiento. Se deberá modificar los tres métodos.
3. Triplicando el coste de trabajo, introduciendo más posibilidad de cometer errores, aumentando la complejidad del código, etc.
4. Si no eres el autor original, la labor de mantenerlo lo complica aún más.

VENTAJAS

Las ventajas de la aplicación del principio son:

- **Hace el código más mantenible**. Cualquier cambio en la funcionalidad lógica, no tienes que cambiarlo en todos los sitios repetidos. Por lo tanto, evita fallos si las funcionalidades repetidas no se encuentran sincronizadas.
- **Reduce el tamaño del código**. Tener menos código ayuda a que el código sea más legible y entendible.
- **Ahorra tiempo**. La disponibilidad de funcionalidades lógicas para reutilizar en el futuro, permite estar más preparados para lograr lo mismo en menos tiempo.

Inversion of Control (IoC)



Como su nombre indica, “inversión de control”, se utiliza en el diseño orientado a objetos para **delegar en un tercero diferentes tipos de flujos de control** para lograr un bajo acoplamiento. Esto incluye el control sobre el flujo de una aplicación y el control sobre el flujo de la creación de un objeto o la creación y vinculación de objetos dependientes.

El principio de IoC ayuda a aumentar la modularidad del programa y al diseño de clases con bajo acoplamiento, lo que las hace testeables, mantenibles y extensibles.

Algunos patrones de diseño son implementaciones de este principio:

- Service locator.
- Dependency injection.
- Template method.
- Strategy.
- Abstract Factory.
- Observer.

A este principio se lo conoce también como *Don’t call us, we’ll call you* (No nos llame, nosotros lo llamamos) o *Hollywood Principle* (Principio de Hollywood).

Inversión de control autentia

¿Qué es?

En la programación tradicional, la interacción entre clases y funciones se hace de forma imperativa. La inversión de control es un principio que delega en un tercero (un framework o contenedor) el control del flujo de un programa para la creación de un objeto, la inyección de objetos dependientes, etc.

¿EN QUÉ CONSISTE?

La inversión de control **está basada en el principio de Hollywood**, ya que era muy habitual la frase que decían los directores a los aspirantes: “No nos llames; nosotros te llamaremos”.

Podemos encontrar distintas formas de implementar la inversión de control. Las formas más conocidas de este principio son:

- Localizador de servicios (Service Locator).
- Inyección de dependencias.

La diferencia fundamental entre ambas es que con el Service Locator las dependencias aún se solicitan **explícitamente** desde la clase dependiente, mientras que con la inyección de dependencias, un agente externo se encarga de proveer las mismas, sin mediar acoplamiento entre la dependencia y su proveedor.

VENTAJAS

- **Reduce el acoplamiento** entre clases y a su vez **aumenta la modularidad y extensibilidad**.
- A raíz del punto anterior, **mejora la testeabilidad** del código ya que al reducir el acoplamiento podemos crear dobles de prueba de las dependencias de una clase de una forma muy sencilla.

Dependencia
CLASS B
CLASS A
Dependencia
CLASS C
IoC
Inyecta
CLASS B
CLASS A
Inyecta
CLASS C

You Aren’t Gonna Need It (YAGNI)

Este principio, que podemos traducir como “No vas a necesitar eso”, es un

principio que indica que **no se deben agregar funcionalidades extras** hasta **que no sea necesario**. La tentación de escribir código que no es necesario pero que puede serlo en un futuro, tiene varias desventajas, como el desperdicio del tiempo que se destinaría para la funcionalidad básica (las nuevas características deben ser depuradas, documentadas y soportadas) o que cuando se requieran las nuevas funcionalidades, estas no funcionen correctamente, ya que hasta que no está definido para qué se puede necesitar, es imposible saber qué debe hacer.

Ver el artículo *El principio YAGNI*, en Adictos al Trabajo, para más detalle.

Keep It Simple, Stupid (KISS)

KISS
E T I T
E M U
P P P
L I
E D
autentia

El principio KISS (🤘💋) es un acrónimo que proviene de la frase inglesa “keep it simple, stupid”, que podemos traducir como “mantenlo simple, estúpido”. Se entiende como la necesidad de minimizar los errores tratando de realizar las tareas de forma efectiva y eficiente complicandose lo mínimo posible.

La simplicidad debe ser un objetivo clave tanto en el diseño, como en el desarrollo de la solución y se debe evitar la **complejidad innecesaria**.

Law of Demeter (LoD)

La Ley de Demeter, también conocida como el *Principle of least knowledge* o principio *Don’t talk to strangers* nos dice que un método de un objeto sólo debería interactuar con:

1. Métodos del propio objeto.
2. Los argumentos que recibe.
3. Cualquier objeto creado dentro del método.

4. Cualquier propiedad / campo directo dentro del propio objeto.

La idea principal es que un objeto **no tiene porqué conocer la estructura interna de los objetos con los que colabora**. En otras palabras, lo que se quiere evitar es el código con una estructura similar a la siguiente:

```
object.getX().getY().getZ().doSomething()
```

¿Cuál es el problema? Básicamente, la cadena denota un fuerte acoplamiento a la estructura de las clases involucradas en ella, afectándonos cualquier cambio o modificación en las mismas.

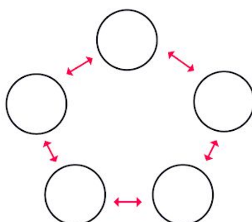
Entre las ventajas de aplicar este principio, encontramos:

- El software resultante tiende a ser más fácil de mantener y adaptar, ya que los objetos dependen menos de la estructura interna de otros objetos, lo que reduce el acoplamiento.
- Se vuelve más sencillo reutilizar las clases.
- El código es más fácil de probar.

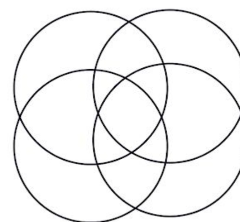
Para más detalle, ver el artículo *Ley de Demeter* en Adictos al Trabajo.

Strive for loosely coupled design between objects that interact

BAJO ACOPLAMIENTO



ALTO ACOPLAMIENTO



autentia

En español “conseguir un diseño débilmente acoplado entre objetos que interactúan”. El acoplamiento se refiere al grado de conocimiento directo que un elemento tiene de otro.

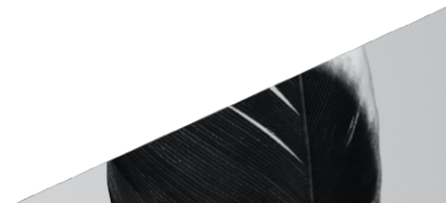
El objetivo es **reducir el riesgo de que un cambio en los objetos con los que interaccionamos provoque cambios en otros objetos**.

Limitar las interconexiones puede ayudar a aislar los problemas cuando las cosas salen mal y simplificar los procedimientos de prueba, mantenimiento y solución de problemas. Esto nos permite construir sistemas flexibles que pueden manejar los cambios porque reducen la dependencia entre múltiples objetos.

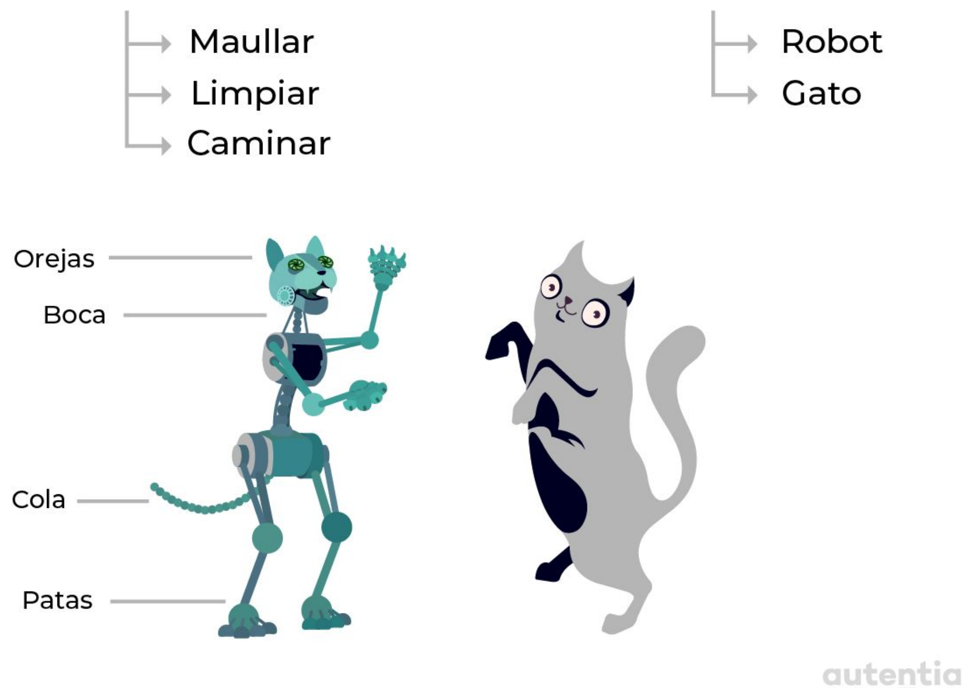
La arquitectura con bajo acoplamiento tiene las siguientes características:

- **Reduce el riesgo** de que un cambio en un elemento pueda provocar cambios en otros elementos.
- **Simplifica las pruebas**, el mantenimiento y la resolución de problemas.
- Los componentes en un sistema débilmente acoplado pueden **reemplazarse** con implementaciones alternativas que brinden los mismos servicios.

Un claro ejemplo de la implementación de este principio es el patrón *Observer*:



Composition over inheritance



El principio de composición sobre la herencia (también conocido como *composite reuse principle*) establece que las clases deben **lograr un comportamiento polimórfico y la reutilización del código mediante la composición** (al contener instancias de otras clases que implementan la funcionalidad deseada), en lugar de a través de la la herencia de una clase base o primaria, siempre que sea posible.

Con la herencia, estructuramos las clases alrededor de lo que son. Con composición, estructuramos las clases basándonos en lo que hacen. Al favorecer la composición sobre la herencia y **pensar en términos de lo que hacen las cosas** en lugar de lo que son, nos liberamos de estructuras de

herencia frágiles y estrechamente acopladas.

El gran problema con la herencia es que tendemos a predecir el futuro, a construir una estructura rígida con un fuerte acoplamiento entre clases padres e hijas en una etapa muy temprana del proyecto y lo más probable, es que cometamos errores de diseño al hacer esto, dado que no podemos predecir el futuro, y cambiar o salir de estas estructuras o taxonomías de herencia es mucho más difícil de lo que parece.

Al favorecer la composición sobre la herencia, dotamos al diseño de una mayor flexibilidad. Es más natural construir clases a partir de varios componentes que tratar de encontrar puntos en común entre ellos y crear un árbol genealógico. Por ejemplo, un pedal acelerador y un volante comparten muy pocos rasgos comunes, sin embargo, ambos son componentes vitales en un automóvil. Lo que pueden hacer y cómo se pueden utilizar para beneficiar al automóvil se define fácilmente. La composición también proporciona un dominio más estable a largo plazo, ya que es menos propenso a las peculiaridades de los miembros. En otras palabras, es mejor componer lo que un objeto puede hacer, verificando que se cumpla la relación *HAS-A* o *TIENE-UN*, que extender lo que es. Esto no significa que nunca se utilice la herencia, se puede implementar siempre que esta sea simple y tenga sentido dentro del modelo y, fundamentalmente, verificando siempre que se cumpla la relación *IS-A* o *ES-UN*.

El diseño inicial se simplifica identificando los comportamientos de los objetos del sistema en interfaces separadas, en lugar de crear una relación jerárquica para distribuir los comportamientos entre las clases a través de la herencia. Este enfoque es más flexible a cambios futuros que de otro modo requerirían una reestructuración completa de las clases de dominio en el modelo de herencia. Además, evita problemas a menudo asociados con cambios relativamente menores en un modelo basado en la herencia

que incluye varias generaciones de clases. Evitando posibles diseños donde se requiera herencia múltiple, ya que muchos lenguajes no la soportan.

Como desventaja, los diseños basados en un enfoque por composición son menos intuitivos.

Encapsulate what varies

Este principio se refiere a que cuando se identifiquen partes de la aplicación que **pueden cambiar**, se deben **aislar** y **encapsular en abstracciones** que permitan realizar el cambio sin afectar a otras partes de la aplicación.

Este principio se apoya en otros dos vistos en apartados anteriores como son *Single responsibility (SRP)* y *Open/closed (OCP)*.

Con la correcta aplicación de este principio se puede obtener dos beneficios fundamentales:

- Cuando una responsabilidad es correctamente acotada en un único módulo, variaciones en los requisitos de esa responsabilidad influyen únicamente en dicho módulo, reduciendo la fragilidad de nuestro sistema y aumentando su reusabilidad.
- La solicitud de nuevos requisitos o nuevos comportamientos se obtiene mediante la incorporación de nuevos elementos en lugar de la modificación de los elementos ya existentes, se reduce la rigidez de nuestro sistema (se vuelve más versátil y flexible) y se reduce también la fragilidad del mismo, ya que el código anterior (y por lo tanto probado) no se modifica.

La mayoría de los patrones de diseño se basan en estos principios. Algunos de estos patrones son:

- Abstract Factory.
- Factory Method.
- Adapter.
- Bridge.
- Decorator.
- Iterator.
- Observer.
- State.
- Strategy.
- Template Method.
- Visitor.

The four rules of simple design

Hay 4 reglas que Kent Beck introdujo en los años 90 sobre los puntos fundamentales que se deben tener en cuenta a la hora de diseñar software, buscando una **manera objetiva** de poder **medir la calidad** de un diseño desde la perspectiva de minimizar los costes y maximizar el beneficio y huyendo de valoraciones subjetivas. Estas cuatro reglas sencillas de recordar están ordenadas por relevancia:

1. **Los tests pasan**: el testing es una pieza que no puede faltar cuando desarrollamos software. El objetivo principal es que cada tarea funcione de la manera esperada y que haya un/unos tests que verifican que estos criterios se cumplen.
2. **Expresan intención**: el código es autoexplicativo, fácil de entender y facilita la comunicación del propósito del mismo.
3. **No hay duplicidades** (DRY): se debe reducir al máximo la duplicidad de la lógica en el código, ya que de no hacerlo así, estaremos construyendo software frágil y cualquier cambio, por muy pequeño que sea, puede “romper” otras partes.

4. **Mínimo número de elementos**: se debe procurar reducir el número de componentes, clases, métodos, etc., a lo imprescindible, eliminando todas aquellas cosas que incrementen la complejidad del sistema de manera innecesaria.

Estas 4 reglas han sido discutidas en gran variedad de libros y foros diversos dando lugar a una buena cantidad de ideas interesantes al respecto, destacando entre otras:

- No hay unanimidad en el orden de los puntos 2 y 3, originando una idea generalizada de que ambos deberían estar en el mismo nivel de importancia.
- El primer punto podría no ser considerado siquiera como un punto del diseño simple, sino como algo esencial y connatural al desarrollo de software. Es decir, ni siquiera se debería plantear la posibilidad de un código sin tests.
- El último punto es considerado para muchos como una consecuencia de la continua aplicación de los puntos 2 y 3.

Las 4 reglas del diseño simple autentia

Origen

Kent Beck describió cuatro reglas fundamentales para diseñar software en la década de los 90. Su objetivo era medir la calidad del software de manera objetiva buscando la perspectiva de minimizar los costes y maximizar el beneficio, huyendo de valoraciones subjetivas.

REGLAS

Las 4 reglas en orden de relevancia son:

1. **Los test pasan**: en el desarrollo del software, el testing nunca debería faltar. Los tests son los garantes de que la tarea funciona como se espera y que los criterios establecidos se cumplen.
2. **Expresan intención**: el código tiene que ser autoexplicativo, sencillo de entender y facilitar la comunicación del propósito del mismo .
3. **Sin duplicaciones** (DRY): debe reducirse al máximo la duplicidad de la lógica en el código. Hacer esto evita construir software frágil.
4. **Mínimo número de elementos**: debe reducirse a lo imprescindible el número de componentes, clases, métodos, etc. eliminando todas aquellas cosas que incrementen la complejidad del sistema de manera innecesaria.

CONTROVERSIA

Tras tanto tiempo desde la definición de estas reglas se han generado multitud de discusiones en foros, libros... dando lugar a cantidad de **ideas interesantes** como:

- No hay unanimidad en el orden de los **puntos 2 y 3**, haciendo a la idea de que ambos están al mismo nivel.
- La **primera regla** podría no ser considerado como una regla del diseño simple, sino como algo esencial o inherente al desarrollo del software. Es decir, no debería plantearse siquiera el desarrollo de código sin tests.
- La **cuarta regla** es considerada para muchos como una consecuencia de la aplicación continua de las reglas 2 y 3.

The boy scout rule

Los Boy Scouts tienen la regla de dejar el campamento más limpio de lo que se lo encontraron y en caso de ensuciarlo, se limpia y se deja lo mejor posible para la siguiente persona que venga. Si se aplica esto al desarrollo de software, se puede decir que si vemos alguna parte del código que se pueda mejorar, independientemente de quién lo haya hecho, debemos hacerlo.

El objetivo principal es mejorar la calidad del código y evitar su deterioro con el fin de ayudar al siguiente desarrollador (o a uno mismo dentro de un tiempo), a cambiar o desarrollar una nueva funcionalidad de una forma más sencilla. Se promueve el trabajo en equipo por encima de la individualidad, ya que no solo es importante la tarea que esa persona ha realizado, sino el proyecto en general y si se ve que algo se puede mejorar, se hace. La idea

es mejorar pequeñas partes de código de manera acotada y segura, ya que tampoco es cuestión de cambiar un módulo entero, sino poco a poco, ir mejorando su calidad.

Para aplicar esta regla, se deben tener claros los principios SOLID.

Para más detalle, ver el artículo *La regla del Boy Scout y la Oxidación del Software* en Adictos al Trabajo.

Last Responsible Moment

El desarrollo de software es una disciplina realmente curiosa. No es extraño, y me atrevería decir que es lo más habitual, encontrarse trabajando ya sobre las funcionalidades de un proyecto incipiente o añadiendo nuevas a otro ya más avanzado, sin tener aún claramente descritos los requisitos. Este principio propone como estrategia para abordar nuestros diseños, diferir nuestras decisiones, especialmente aquellas que se puedan considerar irreversibles, hasta el **último momento posible**. Este momento sería aquel en el que no tomar la decisión, supone un coste mayor que tomarla. Cuanto más tiempo mantengamos nuestras decisiones abiertas, más información iremos acumulando para optar por la decisión más adecuada.

Design Patterns

“Los patrones de diseño son descripciones de objetos y clases conectadas que se personalizan para resolver un problema de diseño general en un contexto particular”.

- Gang of Four

Los patrones de diseño ofrecen **soluciones comunes** a problemas recurrentes de diseño de aplicaciones. En la programación orientada a objetos, los patrones de diseño normalmente están dirigidos a resolver los problemas asociados con la creación e interacción de objetos, en lugar de los problemas a gran escala que afrontan las arquitecturas generales del software. Proporcionan soluciones generalizadas en forma de repeticiones que se pueden aplicar a problemas de la vida real.

Los patrones de diseño son soluciones útiles y probadas para los problemas que inevitablemente aparecen. No solo albergan años de conocimiento y experiencia colectiva, sino que además los patrones de diseño ofrecen un **vocabulario común** entre los desarrolladores y arrojan luz sobre muchos problemas.

Sin embargo, el uso innecesario o excesivo de patrones de diseño puede suponer también una sobre ingeniería, dando como resultado un sistema

excesivamente complejo que lejos de resolver los problemas, los aumenta, dando lugar a un diseño ineficiente, bajo rendimiento y problemas de mantenimiento.

Los patrones de diseño se clasificaron originalmente en tres grupos:

- Creacionales.
- Estructurales.
- De comportamiento.

Con el tiempo, fueron apareciendo nuevos patrones y con ellos, nuevas categorías de problemas que solucionan, por ejemplo, los patrones de concurrencia.

Patrones de diseño GoF autentia

¿Qué son?

Los patrones de diseño **ofrecen soluciones a problemas recurrentes** en el desarrollo del software. Normalmente constan de una serie de pautas a seguir (una receta) que resuelven un problema concreto que ha sido ya probado y documentado por gran parte de la comunidad.

GoF (Gang of Four)

GoF surgió a raíz del libro 'Design Patterns - Elements of Reusable Software' escrito por cuatro desarrolladores que descubren una forma esencial de enfrentarse a la programación.

Aplicando con criterio el uso de patrones, podemos desarrollar **software más robusto, escalable y mantenible**, pero tampoco se debe abusar de ellos y seguirlos al pie de la letra. Debemos ser flexibles ya que dependiendo de nuestras necesidades, se pueden implementar de una forma u otra.

Características de los patrones:

- Proponen **soluciones sólidas a problemas concretos** probadas por la comunidad.
- Buscan **maximizar la cohesión** y **minimizar el acoplamiento**.
- Se basan en principios (SOLID, separation of concerns, ley de Demeter, etc.) que favorecen el **código limpio.**

TIPOS

- **Creacionales:** se utilizan para la instanciación de objetos, encapsulan la lógica de creación y aíslan al cliente de esta responsabilidad. Se intenta evitar el uso del operador ***new*** entre clases para reducir el acoplamiento.
- **Comportamiento:** se utilizan para definir la interacción entre distintos objetos. La interacción debe ser de tal manera que puedan comunicarse fácilmente entre sí, minimizando el grado de acoplamiento.
- **Estructurales:** se utilizan para resolver problemas de composición (agregación) de clases y objetos. Intentan que los cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Normalmente estas relaciones están determinadas por las interfaces que soportan los objetos.

Patrones creacionales

Builder

Este patrón pretende separar la lógica de construcción de su representación. Para ello, define una clase abstracta, Builder, que es la encargada de crear las instancias de los objetos. Los elementos que intervienen son los siguientes:

- Builder: interfaz abstracta que crea los productos.
- Builder concreto: implementación concreta del builder que crea productos de un cierto tipo.
- Director: el encargado de utilizar la clase builder para crear los objetos.



Creacional - Builder autentia

¿En qué consiste?

Patrón creacional que permite la **creación de diferentes representaciones de un objeto.** Se utiliza en situaciones en las que el objeto tiene una gran cantidad de atributos en el constructor por lo que la construcción se realiza en un conjunto de pasos.

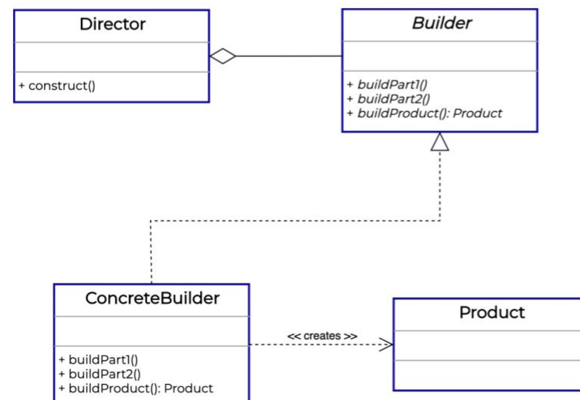


RAZONAMIENTO

Construir un objeto cuyo constructor tiene una larga lista de parámetros, puede llegar a ser tedioso. Sobretodo, porque debemos estar muy pendientes de cuál es el parámetro que va en cada posición. Otro problema es que a veces solo necesitamos construir el objeto con ciertas propiedades ya que las otras no son necesarias. La siguiente línea de código nos muestra un claro ejemplo del problema: **new Product(null, null, null, 'Madrid').** Un objeto Builder soluciona este problema simplificando la construcción y creando un objeto consistente.

También se suele usar este patrón para construir objetos inmutables por lo que la clase del objeto original no debe tener setters. Se tendrá un **'director'** que se encargue de crear siempre los objetos, de este modo, el cliente se abstrae de saber con qué propiedades se está construyendo dicho objeto.

Para el siguiente ejemplo no vamos a tener en cuenta esto, pero debemos conocer otro tipo de variaciones del patrón.





Creacional - Builder autentia

Implementación



SOLUCIÓN

Builder organiza la construcción del objeto en una serie de pasos, siendo estos pasos básicamente los métodos que hay por cada atributo del objeto.

¿Qué ventajas tiene?

- **Mayor control** a la hora de construir un objeto.
- Se pueden construir objetos **inmutables**.

El cliente (Director) dirige la construcción del objeto User usando el UserBuilder:

```
User user = new UserBuilder().city("Madrid").build();
```

```
public class User {
   private String name;
   private String username;
   private Long age;
   private String city;

   public User(String name, String
username,   Long age, String city) {
       this.name = name;
       this.username = username;
       this.age = age;
       this.city = city;
   }
// getters...
}
```

```
public class UserBuilder {
   private String name;
   private String username;
   private Long age;
   private String city;

   public UserBuilder name(String name) {
       this.name = name;
       return this;
   }

   public UserBuilder username(String username) {
       this.username = username;
       return this;
   }

   public UserBuilder age(Long age) {
       this.age = age;
       return this;
   }

   public UserBuilder city(String city) {
       this.city = city;
       return this;
   }

   public User build() {
       return new User(name, username, age, city);
   }
}
```

Singleton

Este patrón consiste en utilizar **una sola instancia** de clase, definiendo así un único punto global de acceso a ella. Dicha instancia es la encargada de la inicialización, creación y acceso a las propiedades de clase.

Este patrón es muy utilizado cuando se quiere controlar el acceso a un único recurso físico (fichero de lectura de uso exclusivo), o haya datos que deban estar disponibles para el resto de objetos de la aplicación (una instancia de log, por ejemplo).

Se define un método de acceso para recuperar la instancia de la clase. Este método también se encargará de crear la instancia en el caso de que se solicite por primera vez. Hay que prestar atención a los problemas que pudiera haber de acceso exclusivo.

Creacional - Singleton autentia

Un único ser sin igual

El patrón *Singleton* resuelve el problema de mantener una única instancia de una clase en memoria durante la ejecución del programa.

DISEÑO

El diseño de este patrón impide que otras clases creen nuevas instancias del Singleton. La primera vez que una clase la necesite, se creará la primera y única instancia de ella.

A partir de ahora, cada vez que se solicite una instancia del Singleton, se hará referencia a la misma instancia, asegurándonos de que no se cree otra.

PRECAUCIÓN

Este patrón se comporta como un objeto global, donde cualquier parte de la aplicación la puede utilizar.

Cambios hechos al estado de una instancia Singleton pueden repercutir en otras clases que la utilicen. Si ocurriese algún problema, es probable que nos resulte difícil de detectar y corregir.

También dificulta el desarrollo de pruebas, ya que el uso de un Singleton implica una dependencia oculta que al momento de hacer pruebas, puede causar sorpresas.

APLICACIÓN

Singleton
- instance: Singleton
- Singleton() + getInstance(): Singleton

A partir del diagrama se infiere que no se podrán crear nuevas instancias de Singleton, dado que el constructor es privado. El método *getInstance* debe ser estático y será el punto de acceso para utilizar la referencia encontrada en *instance*.

UTILIDAD

Un uso común de este patrón se encuentra en componentes que quieren restringir su uso desde un solo punto:

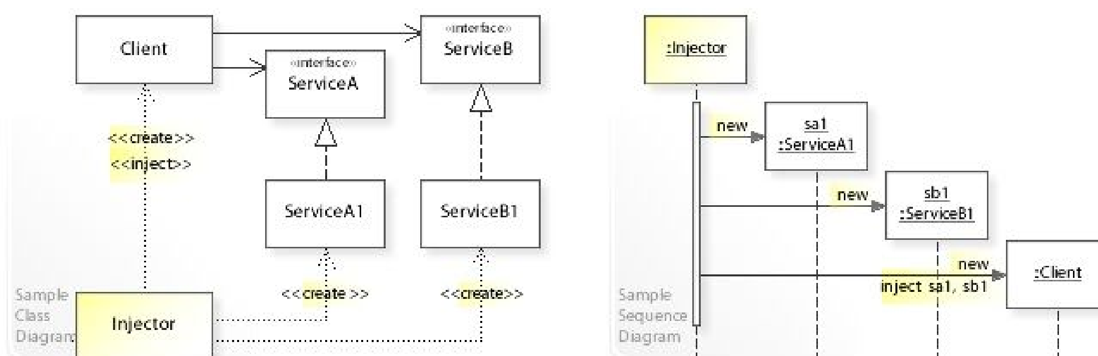
- Parámetros de entorno o de configuración: permite tener una fuente única y rápida de información. Sólo lectura.
- Acceso a interfaces de hardware: se restringe el acceso a recursos que deben ser utilizados uno a la vez y no se pueden paralelizar.

Dependency Injection

Se trata de un patrón de diseño que se encarga de extraer la responsabilidad de la **creación** de instancias de un componente para **delegarla** en otro. Permite que un objeto reciba otros objetos de los que depende, en lugar de ser el propio objeto el que los cree. Estos otros objetos se llaman dependencias. En la típica relación de "uso", el objeto receptor se llama cliente y el objeto pasado (es decir, "inyectado") se llama servicio. El código que pasa el servicio al cliente puede ser de muchos tipos y se llama inyector. En lugar de que el cliente especifique qué servicio usará, el inyector le dice al cliente qué servicio usar. La "inyección" se refiere al paso de una dependencia (un servicio), al objeto (un cliente) que lo usaría.

La inyección de dependencias es una forma de lograr la [inversión de control](#).

El cliente únicamente necesita conocer las interfaces de los servicios sin preocuparse de la implementación real de los mismos



By Vanderjoe - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/wiki/File:W3sDesign_Dependency_Injection_Design_Pattern_UML.jpg

Para más detalle, ver el artículo [Patrón de Inyección de dependencias](#) en Adictos al Trabajo.

Service Locator

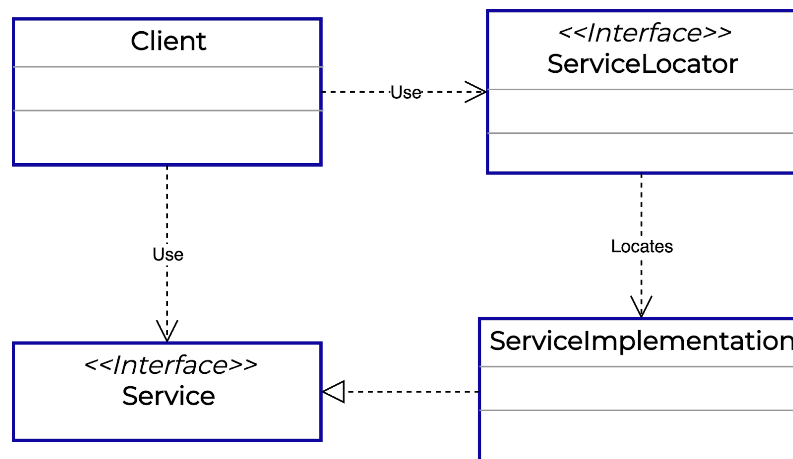
El patrón de localización de servicios es un patrón de diseño utilizado para encapsular los procesos involucrados en la obtención de un servicio con una capa de abstracción fuerte. Este patrón utiliza un registro central conocido como el "Service Locator" que, a demanda, devuelve el componente necesario para realizar una determinada tarea.

Se basa en la creación de una clase, llamada ServiceLocator, que sabe cómo crear las dependencias de otros tipos. A menudo, el localizador de servicios actúa como un depósito para objetos de servicios previamente inicializados. Cuando se requiere uno de estos servicios, se solicita el mismo llamando a un método determinado del ServiceLocator. En algunos casos, el método encargado de la localización de servicios crea instancias de objetos a medida que se necesitan.

Los defensores del patrón dicen que el enfoque simplifica las aplicaciones basadas en componentes, donde todas las dependencias se enumeran limpiamente al comienzo de todo el diseño de la aplicación, lo que hace que la inyección de dependencias tradicional sea una forma más compleja de conectar objetos. Los críticos del patrón argumentan que el software es más difícil de probar.

La principal diferencia frente a la inyección de dependencias es que en este caso hay una solicitud explícita para obtener la dependencia mientras que en la inyección de dependencias la obtención viene ya dada.

Este patrón es otra implementación del principio de inversión de control (IoC).



Abstract Factory

El propósito de Abstract Factory es proporcionar una interfaz para **crear familias** de **objetos** relacionados, sin especificar clases concretas. Normalmente, el cliente crea una implementación concreta de la fábrica abstracta y luego utiliza la interfaz genérica de la misma para crear los objetos concretos. El cliente no sabe (ni le importa) qué objetos concretos obtiene de cada una de estas fábricas internas, ya que utiliza solo las interfaces genéricas de sus productos. Este patrón separa los detalles de la implementación de un conjunto de objetos de su uso general y se basa en la composición del objeto, ya que la creación de objetos se implementa en los métodos expuestos en la interfaz de fábrica.

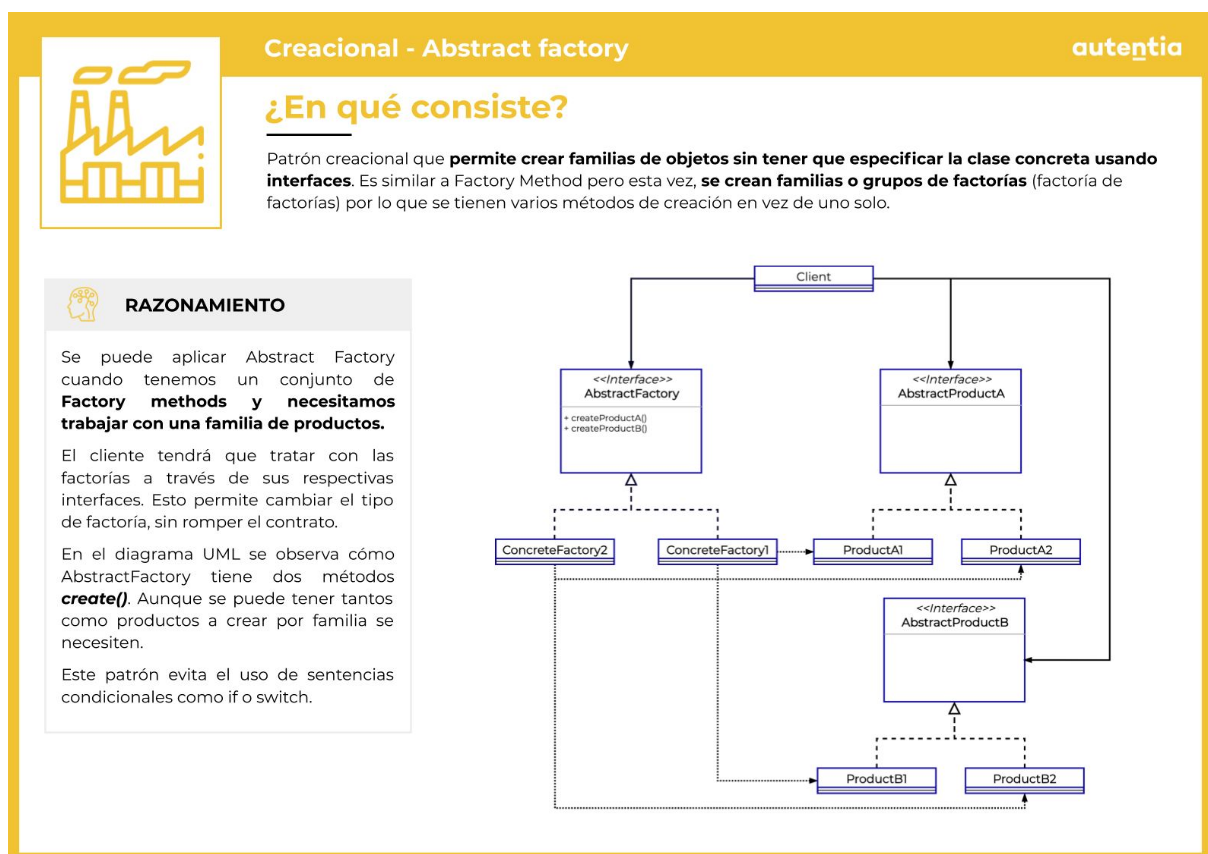
La estructura típica del patrón Abstract Factory es la siguiente:

- Cliente: la clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría.
- Abstract Factory: es la definición de las interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear.
- Factorías Concretas: estas son las diferentes familias de productos. Provee la instancia concreta del tipo de objeto que se encarga de

crear.

- Producto abstracto: definición de las interfaces para la familia de productos genéricos.
- Producto concreto: implementación de los diferentes productos.

Este patrón es otra implementación del principio de inversión de control (IoC).





Creacional - Abstract factory

autentia

Implementación (1/2)



SOLUCIÓN

Tenemos **dos tipos de 'familia'.** La familia Google y la familia Microsoft. Cada familia tiene el producto Mail y el producto CloudStorage, por lo que aplicando **polimorfismo** podemos crear implementaciones específicas para cada producto.

```
public interface CloudStorage {
   String show();
}

public class GoogleCloudStorage implements CloudStorage {
   @Override
   public String show() {
       return "show Google cloud info";
   }
}

public class MicrosoftCloudStorage implements CloudStorage{
   @Override
   public String show() {
       return "show Microsoft cloud info";
   }
}
```

```
public interface Mail {
   String show();
}

public class GoogleMail implements Mail {
   @Override
   public String show() {
       return "show Google mail info";
   }
}

public class MicrosoftMail implements Mail {
   @Override
   public String show() {
       return "show Microsoft mail info";
   }
}
```



Creacional - Abstract factory

autentia

Implementación (2/2)



SOLUCIÓN

La interfaz **AbstractFactory** declara un conjunto de métodos para **crear los productos** (Mail y CloudStorage), pero no sabemos a qué familia pertenecen, solo queremos que mediante esta abstracción, se puedan crear productos (en general), independientemente de si son de una familia u otra.

Dicho esto, la firma de los métodos debe devolver la interfaz correspondiente, de esta manera, el código no se acopla a una única implementación y el cliente se aísla de los detalles.

Se debe crear una implementación de AbstractFactory **por cada familia.**

¿Qué ventajas tiene?

- **Reduce el acoplamiento**.
- Aplica el **Principio de Responsabilidad Unica.**
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación.**

```
public interface AbstractFactory {
   Mail createMail();
   CloudStorage createCloudStorage();
}

public class GoogleFactory implements AbstractFactory {
   @Override
   public Mail createMail() {
       return new GoogleMail();
   }

   @Override
   public CloudStorage createCloudStorage() {
       return new GoogleCloudStorage();
   }
}

public class MicrosoftFactory implements AbstractFactory {
   @Override
   public Mail createMail() {
       return new MicrosoftMail();
   }

   @Override
   public CloudStorage createCloudStorage() {
       return new MicrosoftCloudStorage();
   }
}
```

Factory Method

Provee una interfaz o clase abstracta (creator) que permite encapsular la lógica de creación de los objetos en subclases y éstas deciden qué clase instanciar. Los objetos se crean a partir de un método (factory method) y no a través de un constructor como se hace normalmente. Además, los ConcreteCreators devuelven siempre la interfaz (Product), esto permite que el cliente trate a los productos por igual, tengan una implementación u otra.

La estructura típica del patrón Factory method es la siguiente:

- Product: definición de las interfaces para la familia de productos genéricos.
- ConcreteProduct: implementación de los diferentes productos.
- Creator: declara el factory method que se encargará de instanciar nuevos objetos. Es importante que este método devuelva la interfaz Product. Normalmente el Creator suele ser una clase abstracta con cierta lógica de negocio relacionada con los productos a crear. Dependiendo de la instancia de producto que se devuelva, se puede seguir un flujo u otro.
- ConcreteCreator: crea la instancia del producto concreto.



Creacional - Factory method autentia

¿En qué consiste?

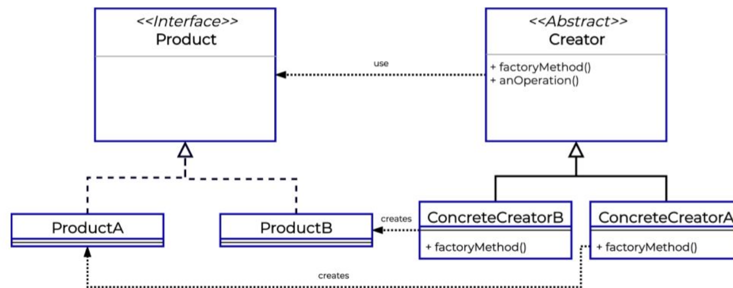
Patrón creacional que provee **una interfaz o clase abstracta**(creator) que permite encapsular la lógica de creación de los objetos en subclases. Las subclases deciden qué clase instanciar. **Los objetos se crean a partir de un método** y no a través de un constructor como se hace normalmente.



RAZONAMIENTO

Al contrario que Simple Factory, que instancia todos los objetos en la misma clase y no hay subclases, **Factory Method crea una implementación o subclase por cada producto**.

En el ejemplo, tenemos como productos distintos tipos de animales. Todos ellos implementan la interfaz ***Animal*** para que siempre se dependa de una abstracción y nunca de una concreción.





Creacional - Factory method autentia

Implementación



SOLUCIÓN

AnimalFactory tiene un método **createAnimal (factory method)** y cada implementación se encarga de la creación de sus objetos. Podríamos incluso crear una factoría **DomesticAnimalFactory** que se encargue de crear solo aquellos animales considerados como domésticos.

Todas las factorías implementan la misma interfaz, por lo que gracias al **polimorfismo** podemos crear tantas implementaciones como necesitemos o cambiar la implementación de una factoría sin que la clase que la use se vea afectada.

¿Qué ventajas tiene?

- **Reduce el acoplamiento y encapsula** el código encargado de crear objetos.
- Aplica el **Principio de Responsabilidad Unica.**
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación.**

Se puede hacer una analogía de este patrón a la programación declarativa. Quiero una instancia del objeto X, pero, cómo se construya por debajo o cómo esté implementado no es mi responsabilidad.

```
public interface Animal {}

public class Cat implements Animal {}

public class Cocodrile implements Animal {}

public class Dog implements Animal {}

public class Lion implements Animal {}


public interface AnimalFactory {
   Animal createAnimal();
}

public class WildAnimalFactory implements AnimalFactory {
   @Override
   public Animal createAnimal() {
       // creates only wild animals
   }
}

public class RandomAnimalFactory implements AnimalFactory {
   @Override
   public Animal createAnimal() {
       // creates random animals
   }
}
```

Patrones estructurales

Adapter

El libro [GoF] indica que este patrón "proporciona una **interfaz unificada** a un conjunto de interfaces en un subsistema". Head First Design Patterns da la misma explicación y señala que convierte la interfaz de una clase en otra interfaz que los clientes esperan. El adaptador permite que las clases puedan trabajar juntas ya que de otro modo, no podrían debido a tener interfaces incompatibles.

En el libro [GoF] se nos describen dos tipos principales de adaptadores:

- Adaptadores de clase: generalmente usan herencia múltiple o varias interfaces para implementarlo.
- Adaptadores de objetos: realizan las composiciones de objetos para adaptarlos.

Un adaptador se puede considerar como la aplicación del principio de inversión de dependencias (DIP), cuando la clase de alto nivel define su propia interfaz (adaptador) para el módulo de bajo nivel (implementado por una clase adaptada).



Estructurales - Adaptador

autentia

¿Qué es?

El patrón adaptador actúa como un **conector entre dos interfaces que son incompatibles y que no pueden estar conectadas directamente**.



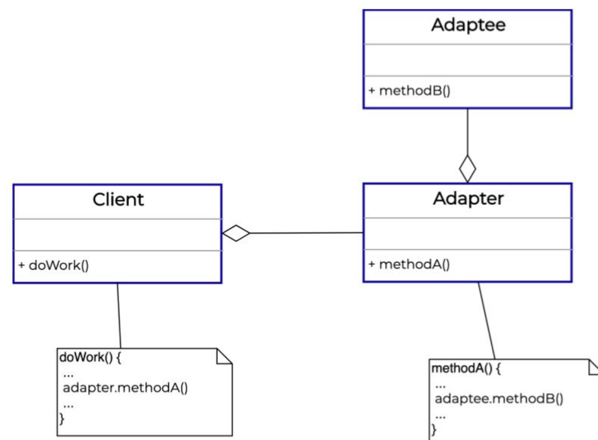
CONCEPTO

Sean dos interfaces A y B incompatibles entre sí existiendo la necesidad de que A llame a B. **Este patrón permite a través de una interfaz adaptador, envolver el contenido de la interfaz B de modo que pueda ser llamada por A**. En el diagrama de la derecha, la clase *Client* interactúa con la clase Adapter para poder utilizar *Adaptee*.

El patrón adaptador es conveniente utilizarlo cuando:

- Un componente tercero ya proporciona una funcionalidad que se puede integrar en nuestro sistema pero es incompatible.
- La aplicación no es compatible con la interfaz que espera consumir el cliente.
- Existe código *legacy* que se quiere utilizar en la aplicación sin tener que hacer ningún cambio sobre él.

Dentro del desarrollo de software este adaptador es también conocido como *wrapper*.





Estructurales - Adaptador

autentia

Implementación



EJEMPLO

Consideremos el escenario en el que tenemos una aplicación diseñada en Estados Unidos que devuelve el precio de un coche en dólares. Queremos utilizar esta funcionalidad dentro de nuestra aplicación pero los precios los tenemos que devolver en euros.

Para ello, vamos a construir una interfaz, *AdapterPricer* que envuelva a la interfaz *Pricer*, que es la que devuelve el precio en dólares.

Las implementaciones de este adaptador tendrán como dependencia la instancia de Pricer que corresponda. Se implementará el método getPrice() de *AdapterPricer* de modo que tras recuperar el precio en dólares de Pricer haga la conversión para devolver el precio en euros.

```
public interface Pricer {
   // Return price in
   double getPrice();
}

public class FerrariPricer implements
Pricer {
   // Return price in $
   double getPrice() {
      return 2000000;
   }
}
```

```
public interface AdapterPricer {
   // Return price in
   double getPrice();
}

public class AdapterPricerImpl implements
AdapterPricer {
   private final Pricer pricer;

   //Constructor

   public double getPrice() {
     ...
     double priceInDolars = pricer.getPrice();
     double priceInEuro =
convertToEuros(priceInDolars);
   }

   private double convertToEuros(double price){
      //Returns price in euros
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      final FerrariPricer ferrariPricer = new FerrariPricer();
      final AdapterPricer ferrariPricerAdapter = new AdapterPricerImpl(ferrariPricer);
      ...
      double ferrariEuroPrice = ferrariPricerAdapter.getPrice();
      // ferrari price in Euros!!
   }
}
```

Data Access Object (DAO)

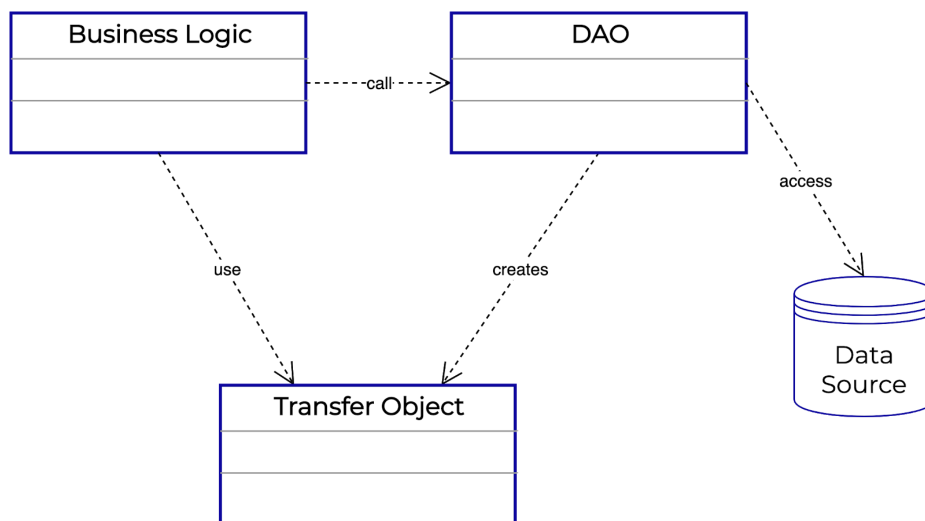
La solución original que propuso el patrón DAO se definió en el libro Core J2EE Patterns: Best Practices and Design Strategies de la siguiente manera:

“Se usa el Data Access Object (DAO) para **abstraer y encapsular todo el acceso a la fuente de datos**. El DAO gestiona la conexión con la fuente de datos para obtener y almacenar datos”.

El problema que se resolvió con la abstracción y la encapsulación de la fuente de datos, fue evitar que la aplicación dependiera de la implementación de la fuente de datos. Esto desacopla la capa de negocio de la fuente de datos.

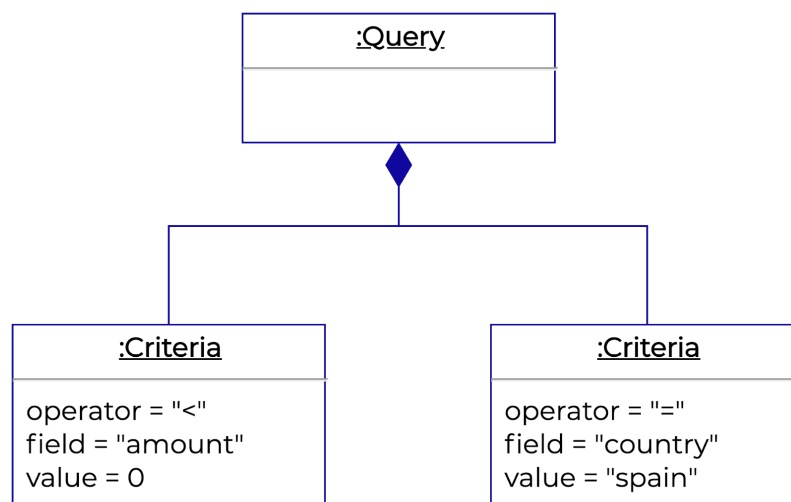
Aunque originariamente permitía protegerse frente a cambios en el motor de base de datos, el patrón DAO sigue siendo un patrón valioso y su solución original, sigue siendo válida. En lugar de protegerse contra el impacto de un cambio improbable en el tipo de fuente de datos, el valor está en su capacidad de prueba y su uso para estructurar el código y mantenerlo limpio de código de acceso a datos.

El patrón DAO encapsula las operaciones de acceso a los datos en una interfaz implementada por una clase en concreto. Si se mockea esta clase, se pueden probar las clases de negocio sin hacer conexiones a la base de datos. La implementación concreta de un DAO utiliza API de bajo nivel para realizar las operaciones de acceso a datos.



Query Object

Este patrón se puede consultar en el libro Patterns of Enterprise Application Architecture. Un Query Object es un **intérprete** [GoF], es decir, una estructura de objetos que puede **formar** una consulta **SQL**. Puede crear esta consulta haciendo referencia a clases y campos en lugar de tablas y columnas. De esta forma, quienes escriben las consultas pueden hacerlo independientemente del esquema de la base de datos y los cambios en el esquema, se pueden localizar en un solo lugar.



Decorator

El propósito de este patrón es el de asignar **responsabilidades adicionales** a un objeto **dinámicamente**, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

La estructura está compuesta por:

- Component: define la interfaz que deben implementar los objetos a los que se les pueden añadir funcionalidades.
- ConcreteComponent: define un objeto al cual se le pueden agregar responsabilidades adicionales. Implementa la interfaz Component.
- Decorator: mantiene una referencia al Component asociado. Implementa la interfaz de la super clase Component, delegando en el Component asociado. El Decorator, en general, añade comportamiento antes o después de un método que ya existe en el Component.
- ConcreteDecorator: añade responsabilidades al Component.

Entre las ventajas de implementar este patrón, podemos encontrar:

- Es más flexible que la herencia.
- Permite añadir y eliminar responsabilidades en tiempo de ejecución.
- Evita la herencia con muchas clases y la herencia múltiple.
- Limita la responsabilidad de los componentes para evitar clases con excesivas responsabilidades en los niveles superiores de la jerarquía.



Estructurales - Decorator autentia

¿En qué consiste?

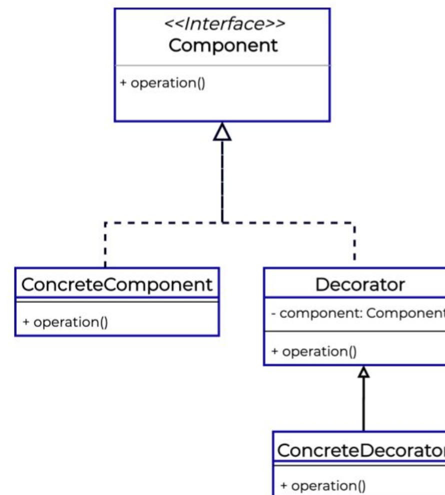
Patrón que **permite añadir nuevas funcionalidades a un objeto en tiempo de ejecución sin modificar su estructura** y a través de una envoltura (wrapper). El decorador envuelve la clase original sin cambiar la firma de los métodos existentes.



CONCEPTO

Decorator ofrece una alternativa cuando no es posible extender el comportamiento de un objeto a través de la herencia.

Normalmente tenemos una interfaz con varias implementaciones. Para aplicar este patrón, debemos crear una nueva 'implementación' que será nuestro *Decorator*. A partir de esta clase, creamos clases concretas de Decorator con las nuevas funcionalidades que se desean añadir.





Estructurales - Decorator autentia

Implementación



SOLUCIÓN

Definimos la interfaz *Text* con su implementación *BaseText* que define un comportamiento básico que podrá ser modificado por un decorator. Importante que la clase *TextDecorator* tenga un campo de tipo Text ya que será el que envolvamos para su posterior modificación. *BoldTextDecorador*, *Italic* y *Underline* definen el comportamiento que se va a añadir dinámicamente al componente, en nuestro caso, a *BaseText*. Desde la perspectiva del cliente, los objetos son iguales.

Ventajas de usar este patrón:

- **Añade o elimina funcionalidades de forma flexible** a un objeto en tiempo de ejecución.
- Sigue el **principio Open/Closed**.
- Permite envolver un objeto en varios decorators.

```
public interface Text {
    void write();
}

public class BaseText implements Text {
    @Override
    public void write() {
        System.out.println("some basic text");
    }
}
```

```
public class TextDecorator implements Text {
    private Text decoratedText;

    public TextDecorator(Text decoratedText) {
        this.decoratedText = decoratedText;
    }

    @Override
    public void write() {
        decoratedText.write();
    }
}
```

```
public class BoldTextDecorator extends
TextDecorator {

    public BoldTextDecorator(Text decoratedText) {
        super(decoratedText);
    }

    @Override
    public void write() {
        System.out.println("adding bold style to
text");
        super.write();
    }
}

/*
 Another class for ItalicTextDecorator...
 Another class for UnderlineTextDecorator...
*/
```

```
public class Main {
    public static void main(String[] args) {
        Text baseText = new BaseText();
        baseText.write();

        Text boldText = new BoldTextDecorator(baseText);
        boldText.write();

        Text italicText = new ItalicTextDecorator(baseText);
        italicText.write();

        //...
    }
}
```

Bridge

Tiene como objetivo **desacoplar** la **abstracción** de la **implementación**. Permite que la abstracción y la implementación se desarrollen de forma independiente a través de un puente (bridge) entre ambas. El código del cliente podrá acceder a la abstracción sin preocuparse por la parte de implementación.

Participantes:

- Abstraction: recibe por parámetro la interfaz que servirá de puente con las implementaciones y es la que se comunicará con el cliente.
- ConcreteAbstraction: puede trabajar con distintas implementaciones a través de la interfaz.
- Implementator: declara una interfaz con sus respectivos métodos que serán los que actúen de enlace entre la abstracción y las implementaciones.
- ConcreteImplementator: contiene código concreto sobre cada implementación.

Entre las ventajas de implementar este patrón podemos encontrar:

- El cliente siempre trabaja con abstracciones y nunca con implementaciones.
- Cumple con el principio Open/Closed ya que se pueden añadir nuevas implementaciones independientes unas de otras.
- Permite reducir el número de subclases que si usaramos herencia pura.



Estructurales - Bridge

autentia

¿En qué consiste?

Es un patrón que busca **desacoplar la abstracción de su implementación**. Permite que la abstracción y la implementación se desarrollen de forma independiente y el código del cliente solo puede acceder a la abstracción sin preocuparse por la parte de implementación. Se puede pensar en una **abstracción con dos capas.**



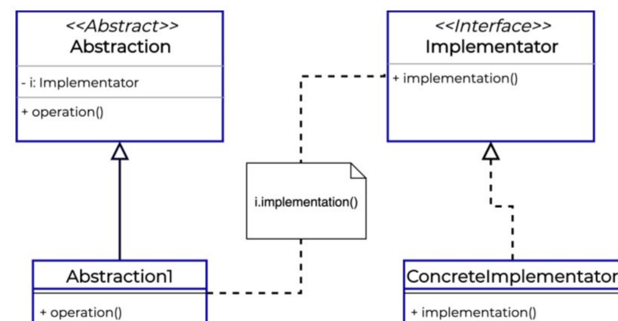
CONCEPTO

Este patrón normalmente define una abstracción y son las implementaciones las que cambian, pero hay casos en los que desde un principio, no estamos seguros del alcance de la abstracción y sabemos que va a ir evolucionando. Para poder **hacer cambios a la abstracción, sin estar rompiendo las implementaciones continuamente,** usamos este patrón.

Este patrón es útil en las siguientes situaciones:

- Queremos que una abstracción y su implementación se definan y extiendan de forma independiente.
- Queremos desacoplar la abstracción y su implementación para poder cambiarla en ejecución.

En la figura se ve la clase Abstraction, que tendrá una referencia a *Implementator*. El método operation() de *Abstraction1* puede trabajar con distintas implementaciones de la interfaz *Implementator*.





Estructurales - Bridge

autentia

Implementación



EJEMPLO

Implementamos la clase abstracta ***RemoteControl*** a la cual podemos inyectar una ***TV***. La ***TV***, en nuestro caso, será el implementator del patrón **Bridge**.

El modo en el que inyectamos el implementator no es relevante para el patrón. Si necesitamos cambiar la implementación en **runtime** podríamos hacerlo mediante un setter en vez de utilizar el constructor.

Tenemos un RemoteControl específico llamado ***ModernRemoteControl*** que tiene la posibilidad adicional de cambiar de canal a la posición siguiente o anterior.

Como ***TV*** podemos tener una digital y otra analógica. De esta forma tenemos **dos jerarquías totalmente independientes** y podemos separar la evolución de ***RemoteControl*** de las implementaciones concretas de ***TV*** para las que se utilizan.

```
public abstract class RemoteControl {

    private final TV implementator;
    private final int currentChannel = 0;

    public RemoteControl(TV implementator) {
        this.implementator = implementator;
    }

    protected final void setChannel(int channel) {
        implementator.setChannel(channel);
        this.currentChannel = channel;
    }

    protected final int getCurrentChannel() {
        return this.currentChannel;
    }
}
```

```
public interface TV {
    setChannel();
}

public class DigitalTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma digital
    }
}

public class AnalogTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma analógica
    }
}
```

```
public class ModernRemoteControl extends RemoteControl {

    public ModernRemoteControl(TV implementator) {
        super(implementator);
    }

    public void previousChannel(int channel) {
        this.setChannel(this.getCurrentChannel() - 1);
    }

    public void nextChannel(int channel) {
        this.setChannel(this.getCurrentChannel() + 1);
    }
}
```

```
public static void main(String[] args) {
    TV implementator = new DigitalTV();
    RemoteControl remoteControl = new ModernRemoteControl(implementator);
    remoteControl.setChannel(5);
}
```

Patrones de comportamiento

Command

El libro The Gang of Four [GoF] indica que se usa el patrón de Comando para "**encapsular** una **solicitud"** como un objeto, permitiendo definir una interfaz común para invocar acciones diversas.

Simplificando, el objetivo de un comando es ejecutar una serie de acciones en su receptor (Receiver). El cliente crea un objeto Command y generalmente, le pasa el Receiver para poder acceder a él. Cuando el Client desea ejecutar el Command, se utiliza el Invoker que almacena el Command y se encarga de iniciar su ejecución en algún momento, invocando al método execute del Command.

Esto permite añadir otras funcionalidades a las acciones como encolamiento, registro, acciones de deshacer o rehacer las operaciones, etc., gracias a **desacoplar** la **solicitud** de una acción de su ejecución.



Comportamiento - Command

autentia

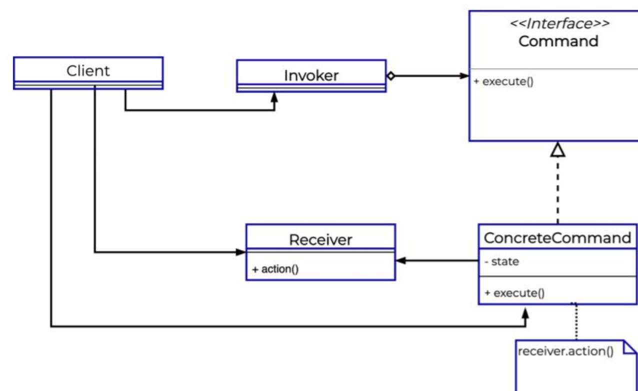
¿En qué consiste?

Patrón que **permite encapsular una petición en un objeto** que contiene toda la información necesaria para realizar una acción (un comando).



CARACTERÍSTICAS

- **Command:** objeto que contiene toda la información necesaria para realizar una acción específica.
- **Receiver:** objeto que realiza las operaciones cuando se ejecuta el comando.
- **Invoker**: objeto encargado de ejecutar la acción, pero desconoce la implementación del comando. Él únicamente recibe la interfaz y llama al método *execute*, aunque también se encarga de gestionar una cola de comandos (en caso de que se necesite) o de realizar la acción de revertir (si fuera necesario). Invoker actúa como mediador y permite desacoplar a los consumidores del objeto comando.
- **Client:** objeto que controla el proceso de ejecución de los comandos. Instancia los comandos deseados y se los pasa al Invoker.





Comportamiento - Command

autentia

Implementación (1/2)



SOLUCIÓN

Se han creado solo dos clases como *Receivers* para simplificar el ejemplo, pero se podrían haber creado más como *Television*, *Radio*, *Heater*, *Cooker* etc., cada una con sus respectivas acciones. Las clases command tienen como atributo su respectivo receiver y se encargan de ejecutar y revertir la acciones pertinentes.

Se podría también, tener una clase TurnOffAllDevicesCommand donde a través del constructor nos llegue una lista de electrodomesticos y se ejecute la acción de apagarlos todos.

Una pequeña desventaja que se puede observar es el incremento del número de clases por comando.

```java
public interface Command {
    void execute();
    void undo();
}
```

```java
//Receiver
public class Speaker {
    public String turnUpVolume() {
        return "turning up volume...";
    }

    public String turnDownVolume() {
        return "turning down volume...";
    }
}
```

```java
//Receiver
public class AirCondition {
    public String on() {
        return "A.C is on";
    }

    public String off() {
        return "A.C is off";
    }
}
```

```java
public class VolumeDownSpeakerCommand
implements Command {
    private Speaker speaker;

    //Constructor

    @Override
    public void execute() {
        speaker.turnDownVolume();
    }

    @Override
    public void undo() {
        speaker.turnUpVolume();
    }
}
//Similar para clase AirConditionOnCommand
```

```java
public class VolumeUpSpeakerCommand
implements Command {
    private Speaker speaker;

    //Constructor

    @Override
    public void execute() {
        speaker.turnUpVolume();
    }

    @Override
    public void undo() {
        speaker.turnDownVolume();
    }
}
// similar para clase AirConditionOffCommand
```

Comportamiento - Command

autentia

Implementación (2/2)

SOLUCIÓN

La clase *RemoteControl* no sabe la implementación del comando que se quiere ejecutar, solo conoce su interfaz.

El cliente será el encargado de instanciar los comandos pertinentes, ya sean tanto para Speaker como para AirCondition, y se los pasará al *Invoker* (RemoteControl).

Algunas ventajas del patrón:

- Cumple con **SRP (Single Responsibility Principle)**. Desacoplamos las clases que invocan las acciones de las que la ejecutan.
- Cumple con el **principio Open Closed**. Se puede introducir tantos comandos como se necesiten.
- Se pueden gestionar operaciones de **reversión o encolado de comandos** (en caso de necesitarlo).

```
//Invoker
public class RemoteControl {
    private Command command;

    public RemoteControl(Command command) {
        this.command = command;
    }

    public void executeOperation(){
        command.execute();
    }

    public void undoOperation(){
        command.undo();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Speaker speaker = new Speaker();
        Command volumeUpSpeaker = new VolumeUpSpeakerCommand(speaker);
        RemoteControl remoteControl = new RemoteControl(volumeUpSpeaker);

        remoteControl.executeOperation();
        remoteControl.undoOperation();

        //...
    }
}
```

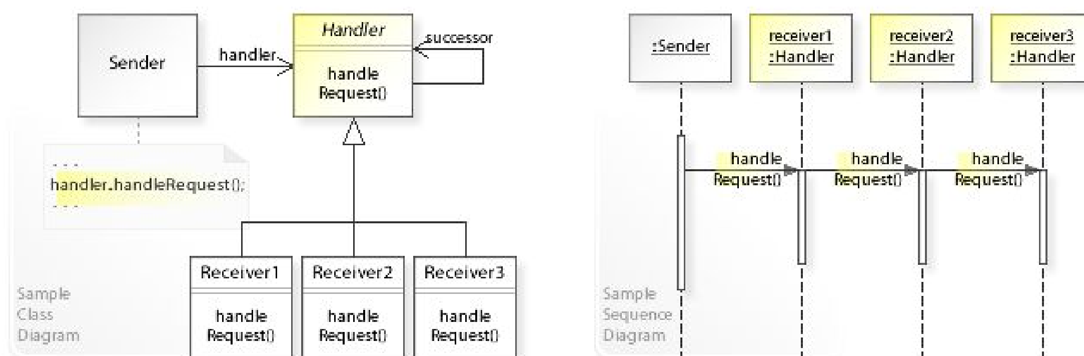
Chain of Responsibility

El libro The Gang of Four [GoF] indica que se usa el patrón “evitar acoplar el remitente de una solicitud a su receptor, al darle a **más de un objeto** la oportunidad de **manejar** la **solicitud**. Se encadenan los objetos receptores y pasa la solicitud a lo largo de la cadena hasta que un receptor lo maneja.”

Aquí se procesan una serie de receptores uno por uno (es decir, de forma secuencial). Una fuente iniciará este procesamiento. Con este patrón, constituimos una cadena donde cada uno de los objetos de receptores puede tener cierta lógica para manejar un tipo particular de objeto. Una vez que se realiza el procesamiento, si aún hay algo pendiente, se puede reenviar al siguiente receptor de la cadena.

Cabe indicar que este tipo de patrón establece una jerarquía entre los receptores, pues los primeros en la cadena tienen prioridad sobre los

siguientes. Podemos agregar nuevos receptores en cualquier momento al final de una cadena.



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=62530468

Este patrón representa una implementación del concepto definido en el principio de responsabilidad única (SRP).

Comportamiento - Chain of Responsibility — autentia



Delegando como se debe

Este patrón permite delegar una operación a lo largo de una cadena de objetos hasta encontrar al eslabón que pueda realizar la acción. Su diseño permite la generación de cadenas dinámicas durante la ejecución del programa, cambiando el comportamiento del componente.



DISEÑO

El diseño de este patrón consiste en definir:

1. Uno o más **eslabones**, o *handlers*. Cada uno tendrá un sólo comportamiento que se ejecutará si las condiciones se cumplen. En caso de no cumplirse, el eslabón tendrá una referencia al siguiente para delegarlo.
2. Un **cliente** inicia la cadena de ejecución.

La construcción de la cadena se puede hacer dentro del cliente o fuera de él, en un componente externo.

La flexibilidad a la hora de generar la cadena permite incluir y/o excluir ciertas condiciones mientras la aplicación se ejecuta.

Podemos controlar el tamaño de la cadena en base a criterios externos (el tipo de mensaje o el tipo de operación que se quiere realizar), abriendo el camino para **reutilizar** los eslabones en otros componentes y/o cadenas.



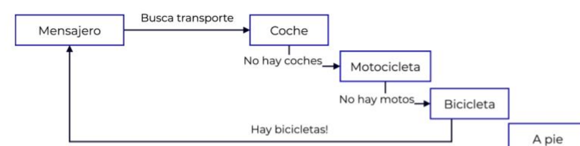
APLICACIÓN

Queremos determinar el medio de transporte para enviar un paquete dentro de una ciudad. Si un medio de transporte no está disponible, se consulta al siguiente. La cadena de prioridad sería la siguiente:

Coche → Motocicleta → Bicicleta → A pie

Obviando la viabilidad del medio de transporte en relación a la distancia, lo que se tiene es una cadena de responsabilidades. Cada eslabón de la cadena sabrá si su medio de transporte se encuentra disponible.

El mensajero utilizará el medio de transporte del primer eslabón que acepte la responsabilidad.



Strategy

El libro The Gang of Four [GoF] indica que se usa este patrón para definir una familia de **algoritmos** que se encapsulan cada uno de forma que sean **intercambiables**. El patrón estrategia permite que el algoritmo varíe independientemente de un cliente a otro.

La clave para aplicar el patrón Strategy es diseñar interfaces para la estrategia y su contexto, que sean lo suficientemente generales como para admitir una variedad de algoritmos. No debería tener que cambiar la estrategia o la interfaz de contexto para admitir un nuevo algoritmo.

Según el patrón de estrategia, los comportamientos de una clase no deben heredarse. En su lugar, deben encapsularse utilizando interfaces. Esto es compatible con el principio Open/Closed (OCP), que propone que las clases deben estar abiertas para la extensión pero cerradas para la modificación.

Para más detalle, ver el artículo Utilizando el patrón Estrategia en Adictos al Trabajo.



Comportamiento - Estrategia

autentia

¿Qué es?

El patrón estrategia define una familia de algoritmos, quedando encapsulados y pudiendo intercambiarse entre ellos. Los algoritmos quedan independientes de los clientes que los usan.



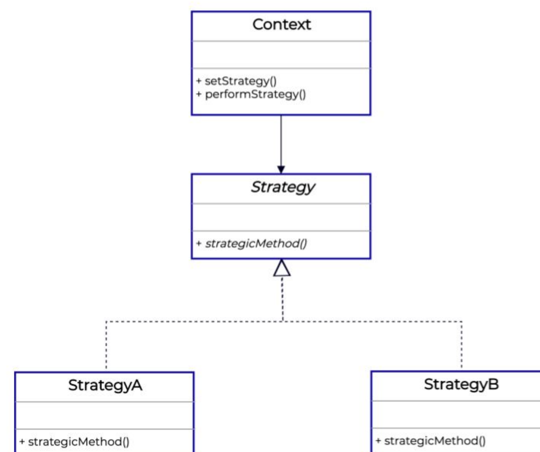
CONCEPTO

Este patrón resulta útil cuando tenemos un problema que se puede resolver de varias formas y queremos la libertad de elegir la forma de hacerlo en ejecución.

Se **define una familia de algoritmos (o estrategias) y el objeto cliente elige entre ellos.** Puede haber cualquier número de estrategias y todas implementan la misma interfaz, por lo que son intercambiables entre sí, incluso en tiempo de ejecución.

De este modo, si hay información del algoritmo que los clientes no deberían saber, ésta va a quedar encapsulada fuera del código del cliente.

Este **patrón es compatible con el principio abierto/cerrado (OCP)**, que propone que las clases deben estar abiertas para extensión y cerradas para modificación. Se añade comportamiento creando nuevas estrategias pero no se modifica el código existente.





Comportamiento - Estrategia

autentia

Implementación



EJEMPLO

Queremos encontrar la ruta para llegar de un punto a otro utilizando los algoritmos de búsqueda de caminos más famosos. Mediante el patrón estrategia vamos a crear la interfaz PathfindingStrategy que en el método **find()** va a hacer los cálculos oportunos para encontrar una ruta.

Tenemos dos implementaciones que extienden la interfaz con los algoritmos más reconocidos para resolver este problema: A* y Dijkstra.

La clase de **Contexto es el cliente de la estrategia**. La estrategia se puede cambiar en cualquier punto de la ejecución, gracias a que tiene un setter disponible.

```
interface PathfindingStrategy {
     public void find();
}

public class AStarAlgorithm implements
PathfindingStrategy {
     public void find() {
          System.out.println("Has usado A*");
     }
}

public class DijkstraAlgorithm implements
PathfindingStrategy {
     public void find() {
          System.out.println("Has usado
Dijkstra");
     }
}
```

```
public class Context {
    PathfindingStrategy c;

    public Context(Strategy c){
        this.c = c;
    }

    public void setStrategy(Strategy c)
{
        this.c = c;
    }

    public void performTask(){
        c.find();
    }
}
```

```
public class Main {
    public static void main(String args[]) {
      // Usamos el algoritmo A*
      PathfindingStrategy aStar = new AStarAlgorithm();
      Context context = new Context(aStar);
      context.performTask();
      // Usamos el algoritmo de Dijkstra
      PathfindingStrategy dijsktra = new DijkstraAlgorithm();
      context.setStrategy(dijkstra);
      context.performTask();
      // Volvemos al algoritmo A*
      context.setStrategy(aStar);
      context.performTask();
    }
}
```

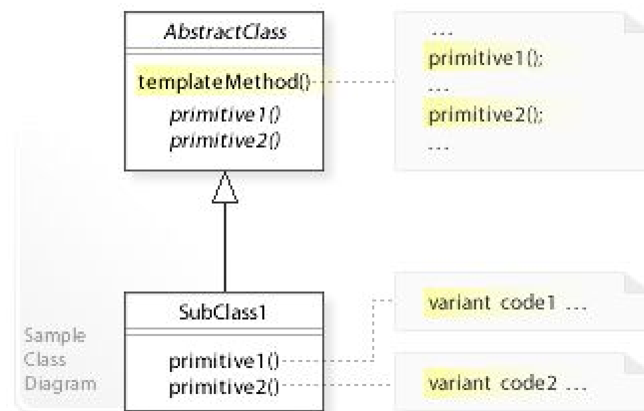
Template Method

El libro The Gang of Four [GoF] indica que se usa este patrón para definir el **esqueleto** de un **algoritmo** en una operación, **delegando** algunos **pasos** a subclases. El método plantilla permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

En un template method, definimos la estructura mínima o esencial de un algoritmo. Luego diferimos algunas funcionalidades (responsabilidades) a las subclases. Como resultado, podemos redefinir ciertos pasos de un algoritmo manteniendo la estructura clave fija para ese algoritmo.

En tiempo de ejecución, el algoritmo representado por el método de plantilla se ejecuta enviando el mensaje de plantilla a una instancia de una de las subclases concretas. A través de la herencia, el método de plantilla en la clase base comienza a ejecutarse delegando parte de los detalles de la implementación en las clases hijas. Este mecanismo garantiza que el algoritmo general siga los mismos pasos cada vez, al tiempo que permite que los detalles de algunos pasos dependan de qué instancia recibió la solicitud original para ejecutar el algoritmo.

Este patrón es un ejemplo de inversión de control porque el código de alto nivel ya no determina qué algoritmos ejecutar, en su lugar, se selecciona un algoritmo de nivel inferior en tiempo de ejecución.



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63155402

Para más detalle, ver el artículo *El patrón de diseño Template Method* en Adictos al Trabajo.

Comportamiento - Template Method autentia

¿En qué consiste?

Este patrón permite **definir el esqueleto de un algoritmo** para luego implementar los detalles del mismo mediante herencia, sin cambiar la estructura del algoritmo.

CARACTERÍSTICAS

El patrón Template Method consiste en definir los pasos de un algoritmo y permitir que **las subclases proporcionen la implementación para uno o más pasos**. De este modo, se pueden definir algunos pasos del algoritmo pero mantener su estructura.

El algoritmo **va a ser un método que dentro invoca a otros métodos en un orden determinado** que son los pasos del algoritmo. Estos pasos serán métodos abstractos que las subclases van a implementar.

Con esto el código está en un solo sitio y además, está protegido dentro de la clase. Añadir nuevas implementaciones del algoritmo solo requiere implementar las operaciones del algoritmo.

También puede haber métodos que no hagan nada, pero que una subclase pueda darles una implementación. A estos métodos se los llama *hooks*.

TEMPLATE METHOD VS STRATEGY

Ambos patrones son usados para encapsular algoritmos, uno mediante herencia y otro mediante composición.

Con el patrón Template Method se define el esqueleto del algoritmo pero se deja a las subclases parte de la responsabilidad. Por otro lado, con el patrón Strategy se define una familia de algoritmos que pueden tener estructuras distintas y se hacen intercambiables en tiempo de ejecución.

```
abstract class AbstractClass {
    final void templateMethod() {
        step1();
        step2();
        concreteOperation();
        hook();
    }
    abstract void step1();
    abstract void step2();

    final void concreteOperation() {
        // implementation here...
    }
    void hook() {}
}
```

Interpreter

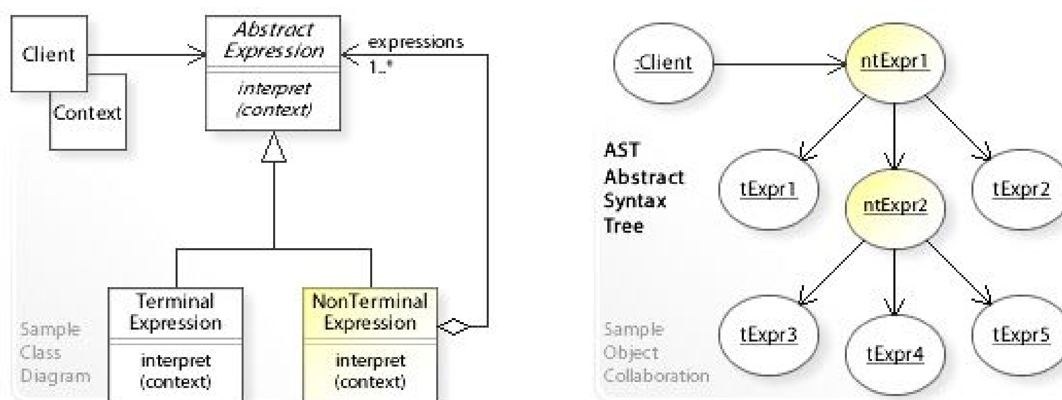
El libro [GoF] lo define de la siguiente manera:

“Dado un lenguaje, define una representación de su **gramática** junto con un **intérprete** que usa dicha representación para interpretar sentencias del lenguaje.”

En otras palabras, el patrón define la gramática de un lenguaje particular de una manera orientada a objetos que puede ser evaluada por el propio intérprete.

Teniendo esto en cuenta, técnicamente podríamos construir nuestra expresión regular personalizada, un intérprete DSL personalizado o podríamos analizar cualquiera de los lenguajes humanos, construir árboles de sintaxis abstracta y luego ejecutar la interpretación.

El patrón intérprete, generalmente debe usarse cuando la gramática es relativamente simple. De lo contrario, podría ser difícil de mantener.



By Vanderjoe - Own work, CC BY-SA 4.0,
https://upload.wikimedia.org/wikipedia/commons/3/33/W3sDesign_Interpreter_Design_Pattern_UML.jpg

Ventajas e inconvenientes:

- Facilidad de cambiar y ampliar: puesto que el patrón usa clases para representar las reglas de la gramática, se puede usar la herencia para cambiar o extender ésta.
- Fácil implementación:. las clases que definen los nodos del árbol

sintáctico abstracto tienen implementaciones similares.

- Las gramáticas complejas son difíciles de mantener ya que define, al menos, una clase para cada regla de la gramática. De ahí que las gramáticas que contienen muchas reglas puedan ser difíciles de controlar y mantener.

Para más detalle, ver el artículo Patrón Intérprete en Adictos al Trabajo.

Observer

Este patrón define una **dependencia** entre objetos de forma que cuando un objeto **cambia** su **estado**, todos los objetos que dependen de él son **notificados** y pueden reaccionar si lo desean a una acción.

El objeto de datos, o Subject, provee de métodos para que cualquier objeto Observer pueda suscribirse o cancelar la suscripción, pasando una referencia de sí mismo al Observable o Subject. El Subject mantiene una lista con las referencias a sus Observers para notificarles cada cambio de estado (si procede).

Los Observers a su vez están obligados a implementar los métodos que utiliza el Subject para notificar a sus Observers de los cambios que sufre para que todos ellos tengan la oportunidad de reaccionar a ese cambio de manera que, cuando se produzca un cambio en el Subject, éste pueda recorrer la lista de Observers notificando a cada uno.

Este patrón es aplicable cuando:

- Una abstracción tiene dos puntos de vista dependientes uno del otro. Encapsular estos puntos de vista en objetos separados permite cambiarlos y reutilizarlos.
- Un cambio en un objeto requiere cambiar otros y no sabemos cuántos objetos van a cambiar.

- Un objeto debería poder notificar a otros sin saber quiénes son.

Entre las ventajas de utilizar este patrón encontramos:

- Podemos modificar el Subject y los Observers de forma independiente al estar desacoplados.
- Nos permite reutilizar tanto Observers como Subjects.
- Respeta el principio Open/Closed, permitiendo añadir Observers sin modificar los Subjects.
- Reduce el acoplamiento entre Subject y Observer, ya que un Subject sólo conoce su lista de Observers a través de un interfaz pero no la clase concreta.
- El Subject se comunica con los Observers mediante broadcast o difusión.

Comportamiento - Observer autentia

Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.

CONCEPTO

Dado el diagrama, tenemos una clase abstracta, **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una manera de implementar el patrón. Se observa que el ConcreteObserver tiene una referencia al ConcreteSubject, lo cual le permite obtener su estado. De esta manera, cuando el Subject notifique a sus Observers, esta implementación en particular, tendrá una referencia directa al Subject para poder reaccionar en base a los cambios sucedidos.

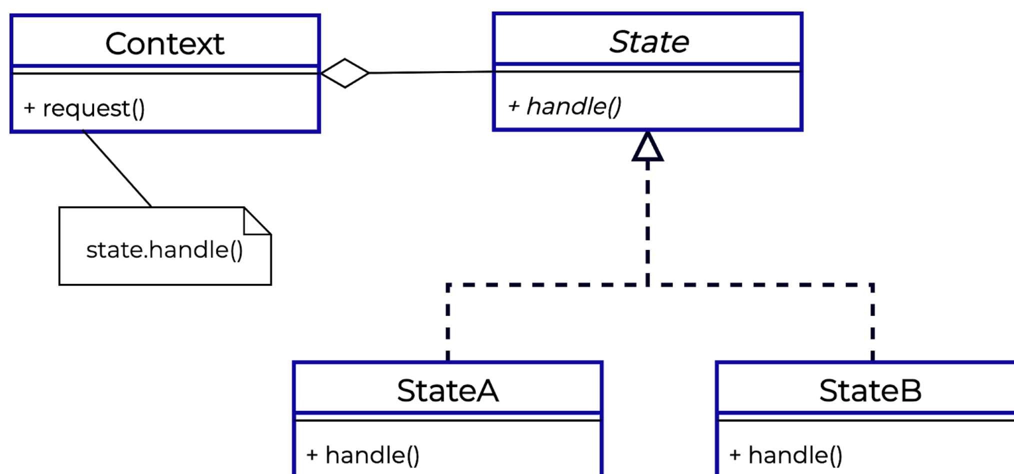
Esta no es la única forma de diseñar el patrón Observer, pero si es la clásica descrita por el *GoF* (Gang of Four).

<<Abstract>>
Subject
+ attach(Observer)
+ detach(Observer)
+ notify()
observers *
<<Interface>>
Observer
+ update(): void
ConcreteSubject
- state: SubjectState
+ setState(SubjectState)
+ getState(): SubjectState
1 subject
ConcreteObserver
- state: ObserverState

State

Según el libro de [GoF], el patrón State “permite que un objeto **modifique** su **comportamiento** cada vez que **cambie** su **estado** interno. Parecerá que cambia la clase del objeto”. Podemos dibujar el comportamiento de nuestro objeto como si se tratase de una “máquina de estados finita”.

El patrón estado se usa para encapsular todo el comportamiento variable de un objeto en función de su estado interno. Es una manera más limpia de construir un objeto que cambia su comportamiento en tiempo de ejecución sin recurrir a declaraciones condicionales, respetando el principio Open/Closed y el Single Responsibility Principle, ya que cada estado está representado por una clase que implementa una interfaz. Esto facilita la mantenibilidad del código.



Participantes:

- Context:
    - Define la interfaz que será usada por los clientes.
    - Mantiene una instancia que representa el estado actual del objeto.
- State:
    - Define una interfaz para encapsular el comportamiento asociado con un determinado estado del Context.
- Subclases de State:
    - Cada subclase implementa un comportamiento asociado a los diferentes estados del Context.

Principales ventajas:

- Separa el comportamiento del objeto por estados. Al aislar el comportamiento en estados y convertirlo en clases separadas, nos permite fácilmente añadir nuevos estados y definir las transiciones hacia ellos.

- Hace explícitas las transiciones entre estados. Podemos evitar las transiciones a estados internos inconsistentes, ya que las transiciones son atómicas para el Context.
- Los objetos State pueden compartirse por diferentes contextos. Siempre que estos no tengan estado interno, comportándose como objetos del patrón Flyweight (sin estado intrínseco y con comportamiento).

Comportamiento - Estado autentia

División del comportamiento con estados

El patrón Estado (o *State*, en inglés) utiliza clases para representar el comportamiento de un objeto en función de su estado.

DISEÑO

El diseño de este patrón consiste primero en definir dos grupos de clases: el contexto y los estados.

- El contexto **representa el estado actual**. Cuando un cliente invoque los métodos del contexto, se utilizará el comportamiento del estado asociado a él.
- Los estados son objetos que implementan una misma interfaz utilizada por el contexto. Cada estado contiene lógica de acuerdo a lo que representa.

También hay que considerar las **transiciones de estado**. Sin transiciones, no habría cambio de comportamiento. Si las transiciones se definen dentro de los estados, deberán modificar el estado actual del contexto.

APLICACIÓN

Supongamos que tenemos un tren, y este tren está siempre encendido:

Train
+ state: State
+ accelerate()
+ brake()
+ openDoors()
+ closeDoors()

State
+ accelerate()
+ brake()
+ openDoors()
+ closeDoors()

Moving
...
Stopped
...
Open
...

1. Podemos **acelerar** el tren. Si el estado actual es **Detenido**, se pasa a **En Marcha**. No se puede acelerar si el tren está **Abierto**.
2. Podemos **frenar** el tren. Si el estado actual es En Marcha y la velocidad del tren llega a 0, el estado pasa a Detenido.
3. Cuando está En marcha, el tren **no podrá abrir o cerrar sus puertas**.
4. Cuando esté Detenido, **podrá abrir las puertas**. En ese caso, pasaría al estado **Abierto.**
5. Del estado Abierto se podrán cerrar las puertas y volvería al estado Detenido.

Con este patrón se pretende que el comportamiento del Contexto (el tren) varíe cuando su estado interno cambie: el objeto Tren se comportará en base a su estado actual.

Visitor

Permite **añadir funcionalidad sin** necesidad de **cambiar** las clases de los elementos en los que va a ejecutarse, a través de los denominados Visitors. El patrón sugiere que situemos el comportamiento en una nueva clase llamada Visitor, en vez de integrarlo todo en la clase base. Cada vez que se

necesite añadir un nuevo comportamiento, se hará en una nueva implementación de un Visitor y la clase base solo tiene que aceptar o delegar este comportamiento en el Visitor correspondiente.

Participantes:

- Visitor: interfaz que declara una serie de métodos, normalmente llamados visit y que reciben como parámetro elementos concretos a los que se les va a añadir funcionalidad. Se deben crear tantos métodos visit como clases concretas tengamos por lo que al llamarse igual, se diferenciarán por la firma del método.
- ConcreteVisitor: implementan la interfaz Visitor y definen las funcionalidades que se aplicarán a los elementos o clase base. Aquí es donde se debe definir el comportamiento que debe tener nuestra clase base. En caso de querer hacer alguna modificación, se hará siempre en los ConcreteVisitors y nunca directamente en el elemento. Recordemos que la clase base únicamente delega al Visitor que realice estas modificaciones.
- Element: interfaz que declara un método que acepta los Visitors.
- ConcreteElement: clase base que implementa la interfaz Element y tiene como objetivo redirigir al Visitor concreto que se encargará de añadir el comportamiento específico.

Comportamiento - Visitor autentia

¿Qué es?

Dada una estructura de objetos compuesta, este patrón **permite añadir funcionalidad sin necesidad de cambiar las clases de los elementos en los que va a ejecutarse**.

CONCEPTO

El patrón Visitor permite extender el comportamiento de las clases de una estructura de objetos, creando una jerarquía de clases separada. Estas nuevas clases son del tipo Visitor y recogen el nuevo comportamiento; **las clases cliente solo tienen que "aceptar" al visitante para delegarle las operaciones.**

Como se ve en la figura, la clase ElementOne no implementa la operación directamente, si no que con el método *accept(visitor)* delega la operación al objeto visitante (que invoca su método *visit()*). Si se quiere añadir una nueva operación, se añade una nueva clase que implemente Visitor, sin tener que modificar ElementoOne o ElementTwo.

La implementación del método *visit()* usada, depende tanto del tipo del elemento como del tipo del Visitor. Esto es lo que se conoce como "double-dispatch".

Este patrón resulta muy útil en los siguientes casos:

- Cuando necesitamos añadir nuevas operaciones frecuentemente.
- Tenemos un mismo algoritmo y queremos que funcione en distintas jerarquías de clases y se encuentre en un solo lugar.
- La estructura de objetos cliente no esperamos que cambie o no es nuestra.

Element
+ accept(visitor:Visitor)
Client
ElementOne
+ accept(visitor:Visitor)
ElementTwo
+ accept(visitor:Visitor)
Visitor
+ visit(element:ElementOne)
+ visit(element:ElementTwo)
VisitorOne
+ visit(element:ElementOne)
+ visit(element:ElementTwo)
VisitorTwo
+ visit(element:ElementOne)
+ visit(element:ElementTwo)

Iterator

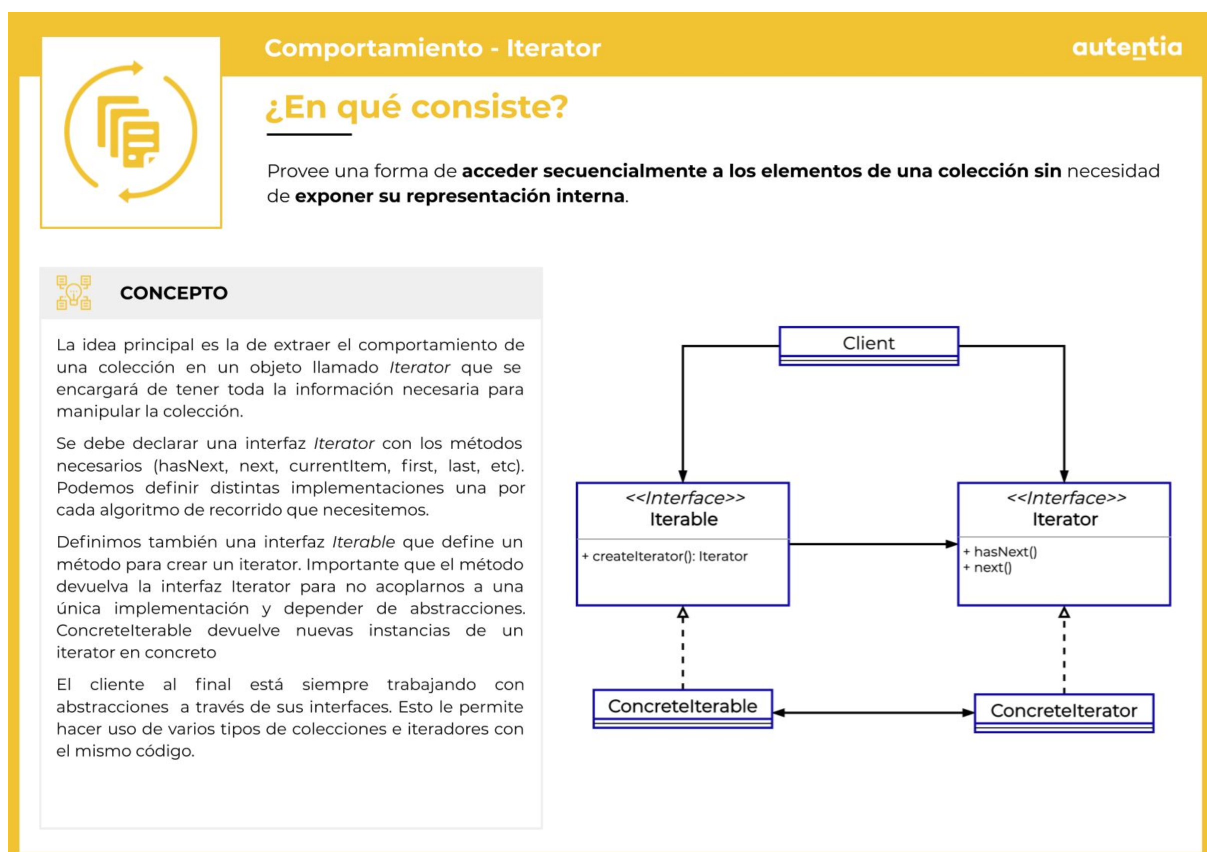
Provee una forma de acceder secuencialmente a los elementos de una colección sin necesidad de exponer su representación interna. El objetivo principal es el de extraer el comportamiento de una colección en un objeto llamado Iterator que se encargará de tener toda la información necesaria para manipularla. El cliente está siempre trabajando con abstracciones a través de sus interfaces. Esto le permite hacer uso de varios tipos de colecciones e iteradores con el mismo código.

Participantes:

- Client: interactúa tanto con Iterator como Iterable a través de sus interfaces para no acoplarse a clases concretas.
- Iterable: declara un método responsable de instanciar el objeto Iterator. Importante que el método devuelva la interfaz para no

acoplarnos a implementaciones.

- ConcreteIterable: implementa la interfaz e instancia el Iterator concreto que iterará sobre una colección específica.
- Iterator: declara los métodos necesarios para recorrer la colección. Puede declarar varios métodos como remove, getFirst, currentItem, size, next, etc.
- ConcreteIterator: implementa los métodos declarados en la interfaz y es responsable de gestionar la posición de la iteración.





Comportamiento - Iterator

autentia

Implementación (1/2)



SOLUCIÓN

Definimos la interfaz Iterator con dos métodos (se podrían añadir más) y su correspondiente implementación. *MessageIterator* debería implementar el algoritmo específico para recorrer esa colección, además de otra lógica, como tener una propiedad que se encargue de gestionar la posición actual de iterator sobre el array/coleccion/lista, etc.

La interfaz Iterable define un método para instanciar iterator. En esta clase podríamos tener una lista con su respectivo método que vaya añadiendo nuevos mensajes y al instanciar *MessageIterator*, se podría pasar dicha lista por parámetro para que pueda ser recorrida posteriormente.

```
public class Message{
  //...
}

public interface Iterable {
   Iterator createIterator();
}

public class MessageIterable
implements Iterable {
   //...

   @Override
   public Iterator createIterator(){
       return new
MessageIterator(messageList);
   }
}
```

```
public interface Iterator {
   Boolean hasNext();
   Object next();
 }

public class MessageIterator implements Iterator{
    private Message[] messageList
    private int currentPosition;
    //constructor

   @Override
   public Boolean hasNext() {
       if(position >= messageList.length){
           return false
        }else {true}
   }

   @Override
   public Object next() {
       Message nextItem = messageList[currentPosition]
       currentPosition += 1;
       return nextItem;
   }
}
```



Comportamiento - Iterator

autentia

Implementación (2/2)



SOLUCIÓN

NotificationScreen será la clase encargada de imprimir los mensajes correspondientes usando los métodos de la clase iterator.

La clase main simplemente define que tipo de iterable desea usar. En este ejemplo, solo tenemos una implementación (*MessageIterable*) pero en un ejemplo real podríamos tener varios tipos.

```
public class NotificationScreen {

   private Iterable messages;

   public NotificationScreen(Iterable messages) {
       this.messages = messages;
   }

   public void showMessages() {
       Iterator iterator = messages.createIterator();
       while (iterator.hasNext()) {
          System.out.println("next item: " + iterator.next());
           //...
       }
   }
}
```

```
public class Main {
   public static void main(String[] args) {
       Iterable messages = new MessageIterable();
       NotificationScreen notificationScreen = new NotificationScreen(mesagges);
       notificationScreen.showMessages();
   }
}
```

Bibliografía

Estas son las fuentes que hemos consultado y en las que nos hemos basado para la redacción de este material:

- The 4 rules of simple design:

  https://blog.jbrains.ca/permalink/the-four-elements-of-simple-design

  https://blog.thecodewhisperer.com/permalink/putting-an-age-old-battle-to-rest

  Understanding the Four Rules of Simple Design, Corey Haines

- The Clean Code Blog by Robert C. Martin (Uncle Bob):

  https://blog.cleancoder.com

- Head First Design Patterns. Eric Freeman & Elisabeth Freeman.
- Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. [GoF]
- Core J2EE Patterns: Best Practices and Design Strategies. Deepak Alur, John Crupi, Dan Malks.
- Patterns of Enterprise Application Architecture. Martin Fowler con la colaboración de David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee y Randy Stafford.

Lecciones aprendidas con esta guía

En esta guía hemos visto los principales principios y patrones de diseño. Estas prácticas no son más que la experiencia de los profesionales del sector, reunidas en una serie de recetas o buenas prácticas para que el **software crezca y evolucione de una forma sostenible.**

Hemos visto los **malos olores** que puede desprender el código y como profesionales, es nuestra responsabilidad irlos solucionando poco a poco y no ir generando una **deuda técnica** que sea como una bola de nieve.

Hacer de **boy scout** va implícito en nuestras tareas del día a día y es lo que va a permitir que las futuras entregas de valor sean rápidas, de calidad y que no rompan funcionalidad existente.

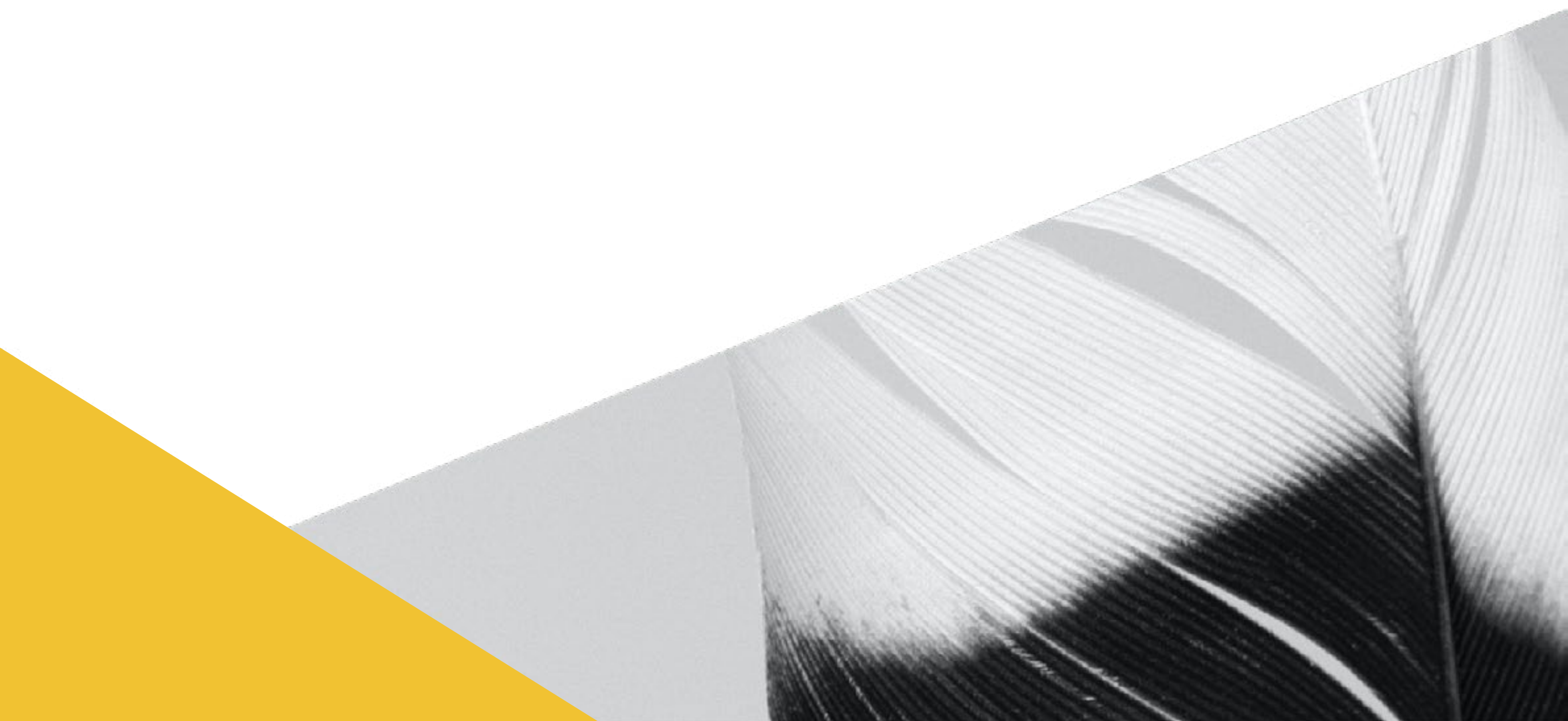
A menudo se pide permiso a managers, product owners o gente de desarrollo de negocio para acometer estas tareas asumiendo que es su decisión, ya que son tareas que conllevan una inversión de tiempo. Esto es un error de base. **La construcción de software es responsabilidad de los desarrolladores de software,** lo mismo que el estado del software.

Conocer los **patrones creacionales, estructurales y de comportamiento** nos van a proporcionar herramientas válidas para entregar valor constante y de calidad.

Conocer las **buenas prácticas y los principios de diseño** nos va a proporcionar las herramientas necesarias para afrontar nuevos desarrollos con seguridad, sabiendo qué decisiones tomar, cuáles aplazar y cómo construir software sobre pilares sólidos, que cambio tras cambio se mantengan firmes.

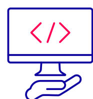
En Autentia proporcionamos soporte al desarrollo de software y ayudamos a la transformación digital de grandes organizaciones siendo referentes en eficacia y buenas prácticas. Te invito a que te informes sobre los servicios profesionales de Autentia y el soporte que podemos proporcionar para la transformación digital de tu empresa.

¡Conoce más!

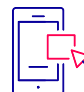


Expertos en creación de software de calidad

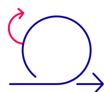
Diseñamos productos digitales y experiencias a medida



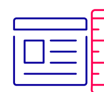
SOPORTE A DESARROLLO
Construimos entornos sólidos para los proyectos, trabajando a diario con los equipos de desarrollo.



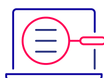
DISEÑO DE PRODUCTO Y UX
Convertimos tus ideas en productos digitales de valor para los usuarios finales.



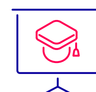
ACOMPAÑAMIENTO AGILE
Ayudamos a escalar modelos ágiles en las organizaciones.



SOFTWARE A MEDIDA
Desarrollamos aplicaciones web y móviles. Fullstack developers, expertos en backend.



AUDITORÍA DE DESARROLLO
Analizamos la calidad técnica de tu producto y te ayudamos a recuperar la productividad perdida.



FORMACIÓN
Formamos empresas, con clases impartidas por desarrolladores profesionales en activo.

www.autentia.com
info@autentia.com | T. 91 675 33 06

¡Síguenos en nuestros canales!







